35th International Symposium on Distributed Computing

DISC 2021, October 4–8, 2021, Freiburg, Germany (Virtual Conference)

Edited by

Seth Gilbert

LIPICS

Editors

Seth Gilbert
National University of Singapore
seth.gilbert@comp.nus.edu.sg

ACM Classification 2012
Software and its engineering → Distributed systems organizing principles; Computing methodologies → Distributed computing methodologies; Computing methodologies → Concurrent computing methodologies; Hardware → Fault tolerance; Networks; Information systems → Data structures; Theory of computation; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-210-5

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-210-5.

Publication date
October, 2021

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.DISC.2021.0

ISBN 978-3-95977-210-5 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Invited Talks

Regular Papers

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Brief Announcements

Preface

Welcome to the DISC 2021, the 35th International Symposium on Distributed Computing, held on October 4–18, 2021. DISC is an international forum on the theory, design, analysis, and implementation of distributed systems and networks, focusing on distributed computing in all its forms. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

This volume contains the papers presented at DISC 2021, including 40 regular papers and 21 brief announcements. Overall, there were 135 papers submitted to DISC on a wide variety of topics in distributed computing. Submissions were double-blind, and they were each reviewed by at least three experts. Final decisions were made during two virtual PC meetings.

This volume also includes the abstracts for two keynote talks, given by Dahlia Malkhi and Bernhard Haeupler. It includes the citations for the best paper and best student paper awards at DISC 2021, as well as citations for two awards jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC):

- The 2021 Edsger W. Dijkstra Prize in Distributed Computing will be presented at DISC 2021 to Paris C. Kanellakis (posthumously) and Scott A. Smolka for their paper "CCS Expressions, Finite State Processes, and Three Problems of Equivalence".
- The 2021 Principles of Distributed Computing Doctoral Dissertation Award will be presented at PODC 2021 to Dr. Leqi Zhu, for his dissertation titled "On the Space Complexity of Colourless Tasks," and to Dr. Goran Zuzic, for his dissertation titled "Towards Universal Optimality in Distributed Optimization."

I would like to thank everyone who contributed to DISC 2021: the authors of the submitted papers, PC members and external reviewers, keynote speakers, members of the organizing committee, workshop organizers, members of the award committees, and participants at the conference. I would also like to thank the members of the steering committee, former chairs and many other members of the community for their valuable assistance and suggestions, EATCS for their support, and the staff at Schloss Dagstuhl – Leibniz-Zentrum für Informatik for their help in preparing these proceedings.

October 2021

Seth Gilbert
DISC 2021 Program Chair

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biannual International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms and WDAG came to stand for International Workshop on Distributed AlGorithms, becoming an annual symposium in 1989. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998, opening the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Chair

Seth Gilbert	National University of Singapore, Singapore

Program Committee

Antonio Fernández Anta	IMDEA Networks Institute
Ittai Abraham	VMware
Dan Alistarh	IST Austria
Naama Ben David	VMware
Michael A. Bender	Stony Brook University
Gregory Chockler	University of Surrey
Michael Dinitz	Johns Hopkins University
Michal Dory	ETH
Yuval Emek	Technion
Seth Gilbert (Chair)	NUS
Rachid Guerraoui	EPFL
Diksha Gupta	NUS
Maurice Herlihy	Brown University
Prasad Jayanti	Dartmouth
Alex Kogan	Oracle Labs
Kuba Łącki	Google Research
Christoph Lenzen	Max-Planck-Institut für Informatik
Alessia Milani	Université de Bordeaux
Dennis Olivetti	University of Freiburg
Rotem Oshman	Tel Aviv University
Ami Paz	University of Vienna
Franck Petit	Sorbonne Université
Peter Robinson	City University of Hong Kong
Gregory Schwartzman	JAIST
Michael L. Scott	University of Rochester
Ilya Sergey	Yale-NUS College
Hsin-Hao Su	Boston College
Lili Su	Northeastern University

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Lewis Tseng	Boston College
Jara Uitto	Aalto University
Jennifer Welch	Texas A&M University
Maxwell Young	Mississippi State University
Leqi Zhu	University of Michigan

Organizing Committee

Alkida Balliu	University of Freiburg
Philipp Bamberger	University of Freiburg
Salwa Faour (Publicity Co-Chair)	University of Freiburg
Marc Fuchs (Publicity Co-Chair)	University of Freiburg
Seth Gilbert (PC Chair)	National University of Singapore
Fabian Kuhn (General Chair)	University of Freiburg
Moti Medina (Workshops/Tutorials Chair)	Bar-Ilan University
Dennis Olivetti	University of Freiburg
Christian Schindelhauer	University of Freiburg
Philipp Schneider	University of Freiburg

Steering Committee

Hagit Attiya	Technion
Seth Gilbert	National University of Singapore
Calvin Newport	Georgetown University
Merav Parter	Weizmann Institute
Andréa Richa (Chair)	Arizona State University
Ulrich Schmid	TU Wien
Jukka Suomela (Vice Chair)	Aalto University

External Reviewers

Vitaly Aksenov
Yackolley Amoussou-Guenou
Sergio Arévalo
Hagit Attiya
Leonid Barenboim
Soheil Behnezhad
Lélia Blin
Borzoo Bonakdarpour
Johannes Bund
Christian Cachin
Mélanie Cambus
Daryus Chandra
Marco Chiesa
Biagio Cosenza
Laxman Dhulipala
Mingzhe Du
Yuval Efron
Chuchu Fan
Angel Alvarez
Pablo Andres-Martinez
James Aspnes
John Augustine
Joao Barreto
Michael Ben-Or
Hans-Joachim Böckenhauer
Quentin Bramas
Janna Burman
Wentao Cai
Armando Castañeda
Yi-Jun Chang
Ashish Choudhury
Peter Davies
Stefan Dobrev
Swan Dubois
Faith Ellen
Wu Feng
Saeed Amiri
Alex Aravind
Sepehr Assadi
Chen Avin
Alan Beadle
Shimon Bitton
Greg Bodwin
Sebastian Brandt
Angela Sara Cacciapuoti
Irina Calciu
Hubert T.H. Chan
Soumyottam Chatterjee
Nachshon Cohen
Stéphane Devismes
Aleksandar Dragojevic
Anaïs Durand
Constantin Enea
Laurent Feuilloley

Orr Fischer
Paola Flocchini
Pierre Fraigniaud
Matthias Függer
Álvaro García Pérez
Vijay Garg
Cyril Gavoille
Peter Gazi
Rati Gelashvili
George Giakkoupis
Yuval Gil
Philip Brighten Godfrey
Wojciech Golab
Alexey Gotsman
Eric Goubault
Jan Friso Groote
Christoph Grunau
Bernhard Haeupler
Joseph Halpern
Nicolas Hanusse
Noga Harlev
David Harris
Qizheng He
Danny Hendler
Juho Hirvonen
Yael Hitron
Damien Imbs
Joseph Izraelevitz
Siddhartha Jayanti
Ernesto Jiménez
Flavio Junqueira
Takashi Katoh
Kimberly Keeton
Seri Khoury
Christoph Kirsch
Lefteris Kokoris-Kogias
Gillat Kol
Christian Konrad
Janne H. Korhonen
Amos Korman
Dariusz Kowalski
Anissa Lamani
Rustam Latypov
Jérémy Ledent
Dean Leitersdorf
Quanquan Liu
Pedro López García
Victor Luchangco
Toshimitsu Masuzawa
Shaked Matar
Yannic Maus
Charles McGuffey
Uri Meir
Slobodan Mitrovic
Neeraj Mittal
Mark Moir
Adam Morrison
William K. Moses Jr.
Achour Mostéfaoui
Doron Mukhtar
Alejandro Naser Pastoriza
Alfredo Navarra
Kartik Nayak
Yasamin Nazari
Calvin Newport
Shreyas Pai
Sreepathi Pai
Roberto Palmieri
Gopal Pandurangan
Dalia Papuc
Matej Pavlovic
Matthieu Perrin
Erez Petrank
Seth Pettie
Cynthia Phillips
Julian Portmann
Nuno Preguiça
Mikaël Rabie
Pedro Ramalhete
Aditya Ramaraju
Lionel Rieg
Václav Rozhoň
Antonio Russo
Fedor Ryabinin
Joel Rybicki
Hamed Saleh
César Sánchez
Nicola Santoro
Vijay Saraswat
TB Schardl
Pierre Sens
Gokarna Sharma
Mark Simkin
Julien Sopena
Suman Sourav
Alexander Spiegelman
Shreyas Srinivas
Chrysoula Stathakopoulou
Yuichi Sudo
Dawei Sun
Yihan Sun
Takeharu Takeharu
Sébastien Tixeuil
Alin Tomescu
Amitabh Trehan
Itay Tsabary
Przemek Uznanski
Giovanni Viglietta
Hoa Vu
Yuanhao Wei
Haosen Wen
Ben Wiederhake
David Wilson
Zhuolun Xiang
Athanasios Xygkis
Igor Zablotchi
Huanchen Zhang
Qinzi Zhang
Jiajia Zhao
Chaodong Zheng
Xiong Zheng

Sponsors

DISC 2021 acknowledges the use of HotCRP for handling submissions and managing the review process, LIPIcs for producing and publishing the proceedings, and Zulip for providing virtual interaction space for conference participants.

DISC thanks VMware for their support.

DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

Awards

Best Paper

The DISC Program Committee has selected the following paper to receive the DISC 2021 best paper award:

Lower Bounds for Shared-Memory Leader Election under Bounded Write Contention
by Dan Alistarh, Giorgi Nadiradze, and Rati Gelashvili.

This paper examines a classical and long-studied problem: electing a leader in a shared memory system. It focuses on the question of how fast a leader election protocol can terminate in a good execution, e.g., when a single process runs all alone. It provides an elegant proof that $\Omega(\log n)$ steps are needed, developing new techniques for proving this type of lower bound. Moreover, the new bound matches the best existing algorithms, showing that the result is tight. As leader election is a foundational problem in distributed computing the new insights in this paper have significant value that merit the best paper award at DISC 2021.

Best Student Paper

The DISC Program Committee has selected the following two papers to receive the DISC 2021 best student paper award:

Broadcast CONGEST Algorithms against Adversarial Edges
by Yael Hitron and Merav Parter.

and

General CONGEST Compilers against Adversarial Edges
by Yael Hitron and Merav Parter.

Both of these papers focus on on a new class of problems in distributed graph theory: algorithms for the adversarial CONGEST model. In the traditional CONGEST model, the network is modeled as a graph where each node can communicate reliably with its neighbors; the key restriction is that nodes can only send a limited amount of information to each neighbor in each round. In the adversarial CONGEST model, by contrast, a subset of the edges are controlled by a malicious adversary that can send arbitrary malicious messages on those edges. The first paper focuses specifically on the task of broadcast, while the second paper develops a general "compiler" that can be used to transform any algorithm into one that is robust to adversarial edge control. For their development of new techniques to design algrotihms for malicious distributed networks, the program committee chose these papers for the best student paper award.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2021 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing is awarded for outstanding papers on the principles of distributed computing, whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade. It is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). The prize is presented annually, with the presentation taking place alternately at PODC and DISC. The committee decided to award the 2021 Edsger W. Dijkstra Prize in Distributed Computing to

Paris C. Kanellakis and Scott A. Smolka

for their paper:

CCS Expressions, Finite State Processes, and Three Problems of Equivalence
Information and Computation, Volume 86, Issue 1, pages 43–68, 1990.

A preliminary version of this paper appeared in the proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC) 1983, pages 228–240.

This paper was a foundational contribution to the fundamental challenge of assigning semantics to concurrent processes, for specification and verification. It addressed the computational complexity of the previously introduced celebrated notion of behavioral equivalence, a cornerstone of Milner's Calculus of Communicating Systems (CCS), aimed at tackling semantics by considering equivalence classes.

With the publication of their PODC 1983 paper, Kanellakis and Smolka pioneered the development of efficient algorithms for deciding behavioral equivalence of concurrent and distributed processes, especially bisimulation equivalence, which is the cornerstone of the process-algebraic approach to modeling and verifying concurrent and distributed systems. Specifically, the main result of their paper is what has come to be known as the K-S Relational Coarsest Partitioning algorithm, which at the time was a new combinatorial problem of independent interest.

The paper also presented complexity results that showed certain behavioral equivalences are computationally intractable. Collectively, Kanellakis and Smolka's results founded the subdiscipline of algorithmic process theory, and helped jump-start the field of Formal Verification.

2021 Award Committee:

Keren Censor-Hillel (chair), Technion
Pierre Fraigniaud, Université de Paris and CNRS
Cyril Gavoille, LaBRI – Université de Bordeaux
Seth Gilbert, National University of Singapore
Andrzej Pelc, Université du Québec en Outaouais
David Peleg, Weizmann Institute of Science

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2021 Principles of Distributed Computing Doctoral Dissertation Award

A pleasingly large number of doctoral dissertations were submitted for the 2021 Principles of Distributed Computing Doctoral Dissertation Award, all of outstanding quality. After careful deliberation, the Committee made the choice to share the award between two theses:

On the Space Complexity of Colourless Tasks
by Leqi Zhu,

and

Towards Universal Optimality in Distributed Optimization
by Goran Zuzic.

Zhu's thesis establishes general memory lower bounds for both deterministic and randomized al- gorithms for a variety of basic synchronization tasks including consensus, k-set agreement, and epsilon-approximate agreement. These bounds hold under a weak liveness assumption – obstruction-freedom – making them very general. Among the results in the thesis one stands out. It provides a definitive solution to a classic and long-standing open problem in distributed computing: to determine the space complexity of consensus in asynchronous, shared-memory systems. Besides the significance of the result, the Committee also appreciated its beautiful execution – a clean, textbook-quality proof. On the basis of this achievement the Committee made its decision to assign the award to this excellent piece of work.

Zuzic's thesis tackles another fundamental problem, in the area of distributed graph algorithms. Loosely speaking, the thesis concerns graph theoretic problems that are non-local, in the sense that they require a number of steps at least proportional to the diameter of the network. This is a large class containing fundamental algorithmic problems such as MST, shortest paths, and min cut. The stated goal is to come up with distributed algorithms that are optimal for every graph topology. In doing so, one must first divine the relevant graph-topology parameters embodying the computational obstruction, and then design algorithms whose performance matches those topological bounds. This is an arduous and ambitious research program, and Zuzic's thesis insightfully covers a lot of ground. For this impressive overall achievement the Committee judged this excellent thesis also worthy of the award.

The 2021 Principles of Distributed Computing Doctoral Dissertation Award Committee:

Marcos K. Aguilera, VMware
Hagit Attiya, Technion
Christian Cachin, University of Bern
Alessandro Panconesi (chair), Sapienza University of Rome

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Quest for Universally-Optimal Distributed Algorithms

Bernhard Haeupler
Carnegie Mellon University, Pittsburgh, PA, USA
ETH Zürich, Switzerland

Abstract

Many distributed optimization algorithms achieve an existentially-optimal round complexity (of $(\tilde{O}(\sqrt{n} + D))$, i.e., there exists some pathological worst-case topology on which no algorithm can be faster. However, most networks of interest allow for exponentially faster algorithms. This motivates two questions:

- What network topology parameters determine the complexity of distributed optimization?
- Are there universally-optimal algorithms that are as fast as possible on every single topology?

This talk provides an overview over the freshly-completed 6-year program that resolves these 25-year-old open problems for a wide class of global network optimization problems including MST, $(1+\epsilon)$-min cut, various approximate shortest path problems, sub-graph connectivity, etc. We provide several equivalent graph parameters that are tight universal lower bounds for the above problems, fully characterizing their inherent complexity. We also give the first universally-optimal algorithms approximately achieving this complexity on every topology.

The quest for universally-optimal distributed algorithms required novel techniques that also answer fundamental (open) questions in seemingly unrelated fields, such as, network information theory, approximation algorithms, (oblivious) packet routing, (algorithmic & topological) graph theory, and metric embeddings. Generally, the problems addressed in these fields explicitly or implicitly ask to jointly optimize ℓ_∞ & ℓ_1 parameters such as congestion & dilation, communication rate & delay, capacities & diameters of subnetworks, or the makespan of packet routings. In particular, results obtained on the way include the following firsts: (Congestion+Dilation)-Competitive Oblivious Routing, Network Coding Gaps for Completion-Times, Hop-Constrained Expanders & Expander Decompositions, Bi-Criteria (Online / Demand-Robust) Approximation Algorithms for many Diameter-Constrained Network Design Problems (e.g., (Group) Steiner Tree/Forest), Makespan-Competitive (Compact and Distributed) Routing Tables, and (Probabilistic) Tree Embeddings for Hop-Constrained Distances.

(Joint work with M. Ghaffari, G. Zuzic, D.E. Hershkowitz, D. Wajc, J. Li, H. Raecke, T. Izumi)

Brief Biography

Bernhard Haeupler's research interests lie in algorithm design, distributed computing, and (network) coding theory. Their research spans over 100 papers and won several awards including the ACM-EATCS Doctoral Dissertation Award of Distributed Computing, a George Sprowls Dissertation Award at MIT, and various best (student) paper awards. Bernhard's research has been funded by multiple prestigious awards including a Sloan Research Fellowship, an NSF Career Award, and an ERC award.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.1

Category Invited Talk

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 1; pp. 1:1–1:1
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Tech Transfer Stories and Takeaways

Dahlia Malkhi
CTO, Diem Association, Geneva, Switzerland

Abstract

In this talk, I will share impressions from several industrial research project experiences that reached production and became part of successful products. I will go through four stories of how these systems transpired and their journey to impact. All of the stories are in the distributed computing arena, and more specifically, they revolve around the state-machine-replication paradigm. Yet, I hope that the take-aways from the experience of building foundations for these systems may be of interest and value to everyone, no matter the discipline.

Brief Biography

Dahlia Malkhi is the Chief Technology Officer at Diem Association, Lead Maintainer of the Diem project, and Lead Researcher at Novi. She has applied and foundational research interests in broad aspects of reliability and security of distributed systems. For over two decades, she has driven innovation in tech, notably as co-inventor of HotStuff, co-founder and technical co-lead of VMware blockchain, co-inventor of Flexible Paxos, the technology behind Log Device, creator and tech lead of CorfuDB, a database-less database driving VMware's NSX-T distributed control plane, and co-inventor of the FairPlay project.

Dahlia Malkhi joined the Diem (Libra) team in June 2019, first as a Lead Reseacher at Novi and later as Chief Technology Officer at the Diem Association. In 2014, after the closing of the Microsoft Research Silicon Valley lab, she co-founded VMware Research and became a Principal Researcher at VMware until June 2019. From 2004-2014, Dahlia Malkhi was a principal researcher at Microsoft Research, Silicon Valley. From 1999-2007, she served as tenured associate professor at the Hebrew University of Jerusalem. From 1995-1999, she was a senior researcher at AT&T Labs, NJ. Dahlia Malkhi holds Ph.D., M.S. and B.S. degrees in computer science from the Hebrew University of Jerusalem.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Tech Transfer, Distributed Systems

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.2

Category Invited Talk

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 2; pp. 2:1–2:1
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Frugal Byzantine Computing

Marcos K. Aguilera
VMware Research, Palo Alto, CA, USA

Naama Ben-David
VMware Research, Palo Alto, CA, USA

Rachid Guerraoui
EPFL, Lausanne, Switzerland

Dalia Papuc
EPFL, Lausanne, Switzerland

Athanasios Xygkis
EPFL, Lausanne, Switzerland

Igor Zablotchi
MIT, Cambridge, MA, USA

Abstract

Traditional techniques for handling Byzantine failures are expensive: digital signatures are too costly, while using $3f+1$ replicas is uneconomical (f denotes the maximum number of Byzantine processes). We seek algorithms that reduce the number of replicas to $2f+1$ and minimize the number of signatures. While the first goal can be achieved in the message-and-memory model, accomplishing the second goal simultaneously is challenging. We first address this challenge for the problem of broadcasting messages reliably. We study two variants of this problem, Consistent Broadcast and Reliable Broadcast, typically considered very close. Perhaps surprisingly, we establish a separation between them in terms of signatures required. In particular, we show that Consistent Broadcast requires at least 1 signature in some execution, while Reliable Broadcast requires $O(n)$ signatures in some execution. We present matching upper bounds for both primitives within constant factors. We then turn to the problem of consensus and argue that this separation matters for solving consensus with Byzantine failures: we present a practical consensus algorithm that uses Consistent Broadcast as its main communication primitive. This algorithm works for $n = 2f+1$ and avoids signatures in the common case – properties that have not been simultaneously achieved previously. Overall, our work approaches Byzantine computing in a frugal manner and motivates the use of Consistent Broadcast – rather than Reliable Broadcast– as a key primitive for reaching agreement.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases Reliable Broadcast, Consistent Broadcast, Consensus, Byzantine Failure, Message-and-memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.3

Related Version *Full Version*: https://arxiv.org/abs/2108.01330 [4]

1 Introduction

Byzantine fault-tolerant computing is notoriously expensive. To tolerate f failures, we typically need $n = 3f + 1$ replica processes. Moreover, the agreement protocols for synchronizing the replicas have a significant latency overhead. Part of the overhead comes from network delays, but digital signatures – often used in Byzantine computing – are even more costly than network delays. For instance, signing a message can be 28 times slower than sending it over a low-latency Infiniband fabric (Appendix A shows the exact measurements).

In this work, we study whether Byzantine computing can be *frugal*, meaning if it can use few processes and few signatures. By Byzantine computing, we mean the classical problems of broadcast and consensus. By frugality, we first mean systems with $n = 2f + 1$ processes, where f is the maximum number of Byzantine processes. Such systems are clearly preferable to systems with $n = 3f + 1$, as they require 33–50% less hardware. However, seminal impossibility results imply that in the standard message-passing model with $n = 2f + 1$ processes, neither consensus nor various forms of broadcast can be solved, even under partial synchrony or randomization [31]. To circumvent the above impossibility results, we consider a message-and-memory (M&M) model, which allows processes to both pass messages and share memory, capturing the latest hardware capabilities of enterprise servers [1, 2]. In this model, it is possible to solve consensus with $n = 2f + 1$ processes and partial synchrony [2].

Frugality for us also means the ability to achieve *low latency*, by minimizing the number of digital signatures used. Mitigating the cost of digital signatures is commonly done by replacing them with more computationally efficient schemes, such as message authentication codes (MACs). For instance, with $n = 3f+1$, the classic PBFT replaces some of its signatures with MACs [21], while Bracha's broadcast algorithm [15] relies exclusively on MACs. As we show, when $n = 2f + 1$, the same signature-saving techniques are no longer applicable.

The two goals – achieving high failure resilience while minimizing the number of signatures – prove challenging when combined. Intuitively, this is because with $n = 2f + 1$ processes, two quorums may intersect only at a Byzantine process; this is not the case with $n = 3f + 1$. Thus, we cannot rely on quorum intersection alone to ensure correctness; we must instead restrict the behavior of Byzantine processes to prevent them from providing inconsistent information to different quorums. Signatures can restrict Byzantine processes from lying, but only if there are enough correct processes to exchange messages and cross-check information. The challenge is to make processes prove that they behave correctly, based on the information they received so far, while using as few signatures as possible.

We focus initially on the problem of broadcasting a message reliably – one of the simplest and most widely used primitives in distributed computing. Here, a designated sender process s would like to send a message to other processes, such that all correct processes deliver the same message. The difficulty is that a Byzantine sender may try to fool correct processes to deliver different messages. Both broadcast variants, Consistent and Reliable Broadcast, ensure that (1) if the sender is correct, then all correct processes deliver its message, and (2) any two correct processes that deliver a message must deliver the *same* message. Reliable Broadcast ensures an additional property: if any correct process delivers a message, then all correct processes deliver that message.

Perhaps surprisingly, in the M&M model we show a large separation between the two broadcasts in terms of the number of signatures (by correct processes) they require. We introduce a special form of indistinguishability argument for $n = 2f + 1$ processes that uses signatures and shared memory in an elaborate way. With it, we prove lower bounds for deterministic algorithms. For Consistent Broadcast, we prove that any solution requires one correct process to sign in some execution, and provide an algorithm that matches this bound. In contrast, for Reliable Broadcast, we show that any solution requires at least $n - f - 2$ correct processes to sign in some execution. We provide an algorithm for Reliable Broadcast based on our Consistent Broadcast algorithm which follows the well-known Init-Echo-Ready pattern [15] and uses up to $n + 1$ signatures, matching the lower bound within a factor of 2.

To lower the impact of signatures on the latency of our broadcast algorithms, we introduce the technique of background signatures. Given the impossibility of completely eliminating signatures, we design our protocols such that signatures are not used in well-behaved executions, i.e., when processes are correct and participate within some timeout. In other words, both broadcast algorithms generate signatures in the background and also incorporate a fast path where signatures are not used.

We next show how to use our Consistent Broadcast algorithm to improve consensus algorithms. The algorithm is based on PBFT [20], and maintains *views* in which one process is the *primary*. Within a view, agreement can be reached by simply having the primary consistent-broadcast a value, and each replicator respond with a consistent broadcast. When changing views, a total of $O(n^2)$ calls to Consistent Broadcast may be issued. The construction within a view is similar to our Reliable Broadcast algorithm. Interestingly, replacing this part with the Reliable Broadcast abstraction does *not* yield a correct algorithm; the stronger abstraction hides information that an implementation based on Consistent Broadcast can leverage. For the correctness of our algorithm, we rely on a technique called *history validation* and on *cross-validating* the view-change message. Our consensus algorithm has four features: (1) it works for $n = 2f + 1$ processes, (2) it issues no signatures on the fast path, (3) it issues $O(n^2)$ signatures on a view-change and (4) it issues $O(n)$ background signatures within a view. As far as we know, no other algorithm achieves all these features simultaneously. This result provides a strong motivation for the use of Consistent Broadcast – rather than Reliable Broadcast – as a first-class primitive in the design of agreement algorithms.

To summarize, we quantify the impossibility of avoiding signatures by proving lower bounds on the number of signatures required to solve the two variants of the broadcast problem – *Consistent* and *Reliable Broadcast* – and provide algorithms that match our lower bounds. Also, we construct a practical consensus algorithm using the Consistent Broadcast primitive. In this work, we consider the message-and-memory model [1, 2], but our results also apply to the pure shared memory model: our algorithms do not require messages so they work under shared memory, while our lower bounds apply *a fortiori* to shared memory.

2 Related Work

Message-and-memory models. We adopt a message-and-memory (M&M) model, which is a generalization of both message-passing and shared-memory. M&M is motivated by enterprise servers with the latest hardware capabilities – such as RDMA, RoCE, Gen-Z, and soon CXL – which allow machines to *both* pass messages and share memory. M&M was introduced by Aguilera et al. in [1], and subsequently studied in several other works [2, 6, 33, 47]. Most of these works did not study Byzantine fault tolerance, but focused on crash-tolerant constructions when memory is shared only by subsets of processes [1, 6, 33, 47]. In [2], Aguilera et al. consider crash- and Byzantine- fault tolerance, as well as bounds on communication rounds on the fast path for a variant of the M&M model with dynamic access permissions and memory failures. However, they did not study any complexity bounds off the fast path, and in particular did not consider the number of signatures such algorithms require.

Byzantine Fault Tolerance. Lamport, Shostak and Pease [40, 46] show that Byzantine agreement can be solved in synchronous message-passing systems iff $n \geq 3f + 1$. In asynchronous systems subject to failures, consensus cannot be solved [32]. However, this result is circumvented by making additional assumptions for liveness, such as randomization [10, 45] or partial synchrony [23, 31]. Even with signatures, asynchronous Byzantine agreement can be solved in message-passing systems only if $n \geq 3f + 1$ [17]. Dolev and Reischuk [30] prove a lower bound of $n(f + 1)/4$ signatures for Byzantine agreement, assuming that every message carries at least the signature of its sender.

Byzantine Broadcast. In the message-passing model, both Consistent and Reliable Broadcast require $n \geq 3f + 1$ processes, unless (1) the system is synchronous and (2) digital signatures are available [17, 29, 50]. Consistent Broadcast is sometimes called Crusader Agreement [29]. The Consistent Broadcast abstraction was used implicitly in early papers on Byzantine broadcast [16, 52], but its name was coined later by Cachin et al. in [19]. The name "consistent broadcast" may also refer to a similar primitive used in synchronous systems [42, 50]. Our Reliable Broadcast algorithm shares Bracha's Init-Echo-Ready structure [15] with other broadcast algorithms [17, 48, 50], but is the first algorithm to use this structure in shared memory to achieve Reliable Broadcast with $n = 2f + 1$ processes.

BFT with stronger communication primitives. Despite the known fault tolerance bounds for asynchronous Byzantine Failure Tolerance (BFT), Byzantine consensus can be solved in asynchronous systems with $2f + 1$ processes if stronger communication mechanisms are assumed. Some prior work solves Byzantine consensus with $2f + 1$ processes using specialized trusted components that Byzantine processes cannot control [24, 25, 26, 27, 34, 53]. These trusted components can be seen as providing a broadcast primitive for communication. These works assume the existence of such primitives as black boxes, and do not study the cost of implementing them using weaker hardware guarantees, as we do in this paper. We achieve the same Byzantine fault-tolerance by using the shared memory to prevent the adversary from partitioning correct processes: once a correct process writes to a register, the adversary cannot prevent another correct process from seeing the written value.

It has been shown that shared memory primitives can be useful in providing BFT if they have *access control lists* or *policies* that dictate the allowable access patterns in an execution [2, 5, 11, 13, 43]. Alon et al. [5] provide tight bounds for the number of strong shared-memory objects needed to solve consensus with optimal resilience. They do not, however, study the number of signatures required.

Early termination. The idea of having a *fast path* that allows early termination in well-behaved executions is not a new one, and has appeared in work on both message-passing [2, 3, 7, 28, 35, 36, 39] and shared-memory [8, 51] systems. Most of these works measure the fast path in terms of the number of message delays (or network rounds trips) they require, but some also consider the number of signatures [7]. In this paper, we show that a signature-free fast path does not prevent an algorithm from having an optimal number of overall signatures.

3 Model and Preliminaries

We consider an asynchronous message-and-memory model, which allows processes to use both message-passing and shared-memory [1]. The system has n processes $\Pi = \{p_1, \ldots, p_n\}$ and a shared *memory* M. Throughout the paper, the term memory refers to M, not to the local state of processes. We sometimes augment the system with eventual synchrony (§3.2).

Communication. The memory consists of single-writer multi-reader (SWMR) read/write atomic registers. Each process can read all registers, and has access to an unlimited supply of registers it can write. If a process p can write to a register r, we say that p *owns* r. This model is a special case of access control lists (ACLs) [43], and of dynamically permissioned memory [2]. Additionally, every pair of processes p and q can send messages to each other over links that satisfy the *integrity* and *no-loss* properties. Integrity requires that a message m from p be received by q at most once and only if m was previously sent by p to q. No-loss requires that a message m sent from p to q be eventually received by q.

Signatures. Our algorithms assume digital signatures: each process can *sign* and *verify* signatures. A process p may sign a value v, producing $\sigma_{p,v}$; when unambiguous, we drop the subscripts. Given v and $\sigma_{p,v}$, a process can verify whether $\sigma_{p,v}$ is a valid signature of v by p.

Failures. Up to f processes may fail by becoming Byzantine, where $n = 2f + 1$. Such a process can deviate arbitrarily from the algorithm, but cannot write on a register that is not its own, and cannot forge the signature of a correct process. As usual, Byzantine processes can collude, e.g., by using side-channels to communicate. The memory M does not fail; such a reliable memory is implementable from a collection of fail-prone memories [2]. We assume that these individual memories may only fail by crashing.

3.1 Broadcast

We consider two broadcast variants: Consistent Broadcast [18, 19] and Reliable Broadcast [14, 18]. In both variants, broadcast is defined in terms of two primitives: *broadcast*(m) and *deliver*(m). A designated *sender* process s is the only one that can invoke *broadcast*. When s invokes *broadcast*(m) we say that s *broadcasts* m. When a process p invokes *deliver*(m), we say that p *delivers* m.

▶ **Definition 3.1.** Consistent Broadcast *has the following properties:*

Validity *If a correct process s broadcasts m, then every correct process eventually delivers m.*

No duplication *Every correct process delivers at most one message.*

Consistency *If p and p' are correct processes, p delivers m, and p' delivers m', then m=m'.*

Integrity *If some correct process delivers m and s is correct, then s previously broadcast m.*

▶ **Definition 3.2.** Reliable Broadcast *has the following properties:*

Validity, No duplication, Consistency, Integrity *Same properties as in Definition 3.1.*

Totality *If some correct process delivers m, then every correct process eventually delivers a message.*

We remark that both broadcast variants behave the same way when the sender is correct and broadcasts m. However, when the sender is faulty Consistent Broadcast has no delivery guarantees for correct processes, i.e., some correct processes may deliver m, others may not. In contrast, Reliable Broadcast forces every correct process to eventually deliver m as soon as one correct process delivers m.

3.2 Consensus

▶ **Definition 3.3.** Weak Byzantine agreement [37] *has the following properties:*

Agreement *If correct processes i and j decide val and val', respectively, then $val = val'$.*

Weak validity *If all processes are correct and some process decides val, then val is the input of some process.*

Integrity *No correct process decides twice.*

Termination *Eventually every correct process decides.*

Our consensus algorithm (§6) satisfies agreement, validity, and integrity under asynchrony, but requires eventual synchrony for termination. That is, we assume that for each execution there exists a *Global Stabilization Time (GST)*, unknown to the processes, such that from GST onwards there is a known bound Δ on communication and processing delays.

4 Lower Bounds on Broadcast Algorithms

We show lower bounds on the number of signatures required to solve Consistent and Reliable Broadcast with $n = 2f+1$ processes in our model. We focus on signatures by correct processes because Byzantine processes can behave arbitrarily (including signing in any execution).

4.1 High-Level Approach

Broadly, we use indistinguishability arguments that create executions E_v and E_w that deliver different messages v and w; then we create a composite execution E where a correct process cannot distinguish E from E_v, while another correct process cannot distinguish E from E_w, so they deliver different values, a contradiction. Such arguments are common in message-passing system, where the adversary can prevent communication by delaying messages between correct processes. However, it is not obvious how to construct this argument in shared memory, as the adversary cannot prevent communication via the shared memory, especially when using single-writer registers that cannot be overwritten by the adversary. Specifically, if correct processes write their values and read all registers, then for any two correct processes, at least one sees the value written by the other [9]. So, when creating execution E in which, say E_v occurs first, processes executing E_w will know that others executed E_v beforehand.

We handle this complication in two ways, depending on whether the sender signs its broadcast message. If the sender does not sign, we argue that processes executing E_w cannot tell whether E_v was executed by correct or Byzantine processes, and must therefore still output their original value w. This is the approach in the lower bound proof for Consistent Broadcast (Lemma 4.1).

However, once a signature is produced, processes can save it in their memory to prove to others that they observed a valid signature. Thus, if the sender signs its value, then processes executing E_w cannot be easily fooled; if they see two different values signed by the sender, then the sender is provably faulty, and correct processes can choose a different output. So, we need another way to get indistinguishable executions. We rely on a *correct bystander* process. We make a correct process b in E sleep until all other correct processes decide. Then b wakes up and observes that E is a composition of E_v and E_w. While b can recognize that E_v or E_w was executed by Byzantine processes, it cannot distinguish which one. So, b cannot reliably output the same value as other correct processes. We use this construction for Reliable Broadcast, but we believe it applies to other agreement problems in which all correct processes must decide.

The proof is still not immediate from here. In particular, since $f < n/2$, correct processes can wait until at least $f+1$ processes participate in each of E_v and E_w. Of those, in our proof we assume at most $f-1$ processes sign values. Since we need a bystander later, only $2f$ processes can participate. Thus, the sets executing E_v and E_w overlap at two processes; one must be the sender, to force decisions in both executions. Let p be the other process and S_v and S_w be the set that execute E_v and E_w respectively, without the sender and p. Thus, $|S_v| = |S_w| = f-1$.

The key complication is that if p signs its values in one of these two executions, we cannot compose them into an execution E in which the bystander b cannot distinguish which value it should decide. To see this, assume without loss of generality that p signs a value in execution E_w. To create E, we need the sender s and the set S_w to be Byzantine. The sender will produce signed versions of both v and w for the two sets to use, and S_w will pretend to execute E_w even though they observed that E_v was executed first. Since $|S_w| + |\{s\}| = f$, all other processes must be correct. In particular, p will be correct, and will not produce the signature that it produces in E_w. Thus, the bystander b will know that S_v were correct. More generally, the problem is that, while we know that at most $f-1$ processes sign, we do not know *which* processes sign. A clever algorithm can choose signing processes to defeat the indistinguishability argument – in our case, this happens if p is a process that signs.

Due to this issue, we take a slightly different approach for the Reliable Broadcast lower bound, first using the bystander construction to show that any Reliable Broadcast algorithm must produce *a single non-sender* signature. To strengthen this to our bound, we construct an execution in which this signature needs to be repeatedly produced. To make this approach work, we show not just that *there exists* an execution in which a non-sender signature is produced, but that *for all* executions of a certain form, a non-sender signature is produced. This change in quantifiers requires care in the indistinguishability proof, and allows us to repeatedly apply the result to construct a single execution that produces many signatures.

4.2 Proofs

In all proofs in this section, we denote by s the designated sender process in the broadcast protocols we consider. We first show that Consistent Broadcast requires at least one signature.

▶ **Lemma 4.1.** *Any algorithm for Consistent Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 1$ has an execution in which at least one correct process signs.*

Proof. By contradiction, assume there is some algorithm A for Consistent Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 1$ without any correct process signing. Partition Π into 3 subsets: S_1, S_2, and $\{p\}$, where S_1 contains the sender, $|S_1| = f$, $|S_2| = f$, and p is a single process. Let v, w be two distinct messages. Consider the following executions.

Execution $E_{\text{clean-v}}$. Processes in S_1 and p are correct (including the sender s), while processes in S_2 are faulty and never take a step. Initially, s broadcasts v. Since s is correct, processes in S_1 and p eventually deliver v. By our assumption that correct processes never sign, processes in S_1 and p do not sign in this execution; processes in S_2 do not sign either, because they do not take any steps.

Execution $E_{\text{dirty-w}}$. Processes in S_1 and S_2 are correct but p is Byzantine. Initially, p sends all messages and writes to shared memory as it did in $E_{\text{clean-v}}$ (it does so without following its algorithm; p is able to do this since no process signed in $E_{\text{clean-v}}$). Then, the correct sender s broadcasts w and processes in S_1 and S_2 execute normally, while p stops executing. Then, by correctness of the algorithm, eventually all correct processes deliver w. By our assumption that correct processes never sign, processes in S_1 and S_2 do not sign in this execution; p does not sign either, because it acts as it did in $E_{\text{clean-v}}$.

Execution E_{bad}. Processes in S_1 are Byzantine, while processes in S_2 and p are correct. Initially, processes in S_2 sleep, while processes in S_1 and p execute, where processes in S_1 send the same messages to p and write the same values to shared memory as in $E_{\text{clean-v}}$ (but they do not send any messages to S_2), so that from p's perspective the execution is indistinguishable from $E_{\text{clean-v}}$. S_1 are able to do this because no process signed in $E_{\text{clean-v}}$. Therefore, p eventually delivers v. Next, processes in S_1 write the initial values to their registers[1]. Now, process p stops executing, while processes in S_1 and S_2 execute the same steps as in $E_{\text{dirty-w}}$ – here, note that S_2 just follows algorithm A while S_1 is Byzantine and pretends to be in an execution where s broadcasts w (S_1 is able to do this because no process

[1] Recall that registers are single-writer. By "their registers", we mean the registers to which the processes can write.

signed in $E_{\text{DIRTY-W}}$). Because this execution is indistinguishable from $E_{\text{DIRTY-W}}$ to processes in S_2, they eventually deliver w. At this point, correct process p has delivered v while processes in S_2 (which are correct) have delivered w, which contradicts the consistency property of Consistent Broadcast. ◀

An algorithm for Reliable Broadcast works for Consistent Broadcast, so Lemma 4.1 also applies to Reliable Broadcast.

We now show a separation between Consistent Broadcast and Reliable Broadcast: any algorithm for Reliable Broadcast has an execution where at least $f-1$ correct processes sign.

The proof for the Reliable Broadcast lower bound has two parts. First, we show that intuitively there are many executions in which some process produces a signature: if E is an execution in which (1) two processes never take steps, (2) the sender is correct, and (3) processes fail only by crashing, then some non-sender process signs. This is the heart of the proof, and relies on the indistinguishability arguments discussed in Section 4.1. Here, we focus only on algorithms in which at most f correct processes sign, otherwise the algorithm trivially satisfies our final theorem.

▶ **Lemma 4.2.** *Let A be an algorithm for Reliable Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 2$ processes, such that in any execution at most f correct processes sign. In all executions of A in which at least 2 processes crash initially, processes fail only by crashing, and the sender is correct, at least one correct non-sender process signs.*

Proof. By contradiction, assume some algorithm A satisfies the conditions of the lemma, but there is some execution of A where the sender s is correct, processes fail only by crashing, and at least 2 processes crash initially, but no correct non-sender process signs. Let $E_{\text{CLEAN-V}}$ be such an execution, D be a set with two processes that crash initially in $E_{\text{CLEAN-V}}$[2], $C = \Pi \setminus D$, and v be the message broadcast by s in $E_{\text{CLEAN-V}}$. Consider the following executions:

EXECUTION $E_{\text{CLEAN-W}}$. The sender s broadcasts some message $w \neq v$, D crashes initially, and C is correct. Since s is correct, eventually all correct processes deliver w. By assumption, at most f processes sign. Let $S \subset C$ contain all processes that sign, augmented with any other processes so that $|S| = f$. Let $T = C \setminus S$. Note that (1) $|T| = f - 1$ and (2) if s signed, then $s \in S$, otherwise $s \in T$.

EXECUTION $E_{\text{CLEAN-V}}$. This execution was defined above (where s broadcasts v). Since s is correct, eventually all correct processes deliver v. At least one process in T is correct – call it p_t – since processes in D are faulty and there are at least $f + 1$ correct processes. Note that p_t delivers v. We refer to p_t in the next execution.

EXECUTION $E_{\text{MIXED-V}}$. Processes in S are Byzantine and the rest are correct. Initially, the execution is identical to $E_{\text{CLEAN-V}}$, except that (1) processes in D are just sleeping not crashed, and (2) processes in S do not send messages to processes in D (this is possible because processes in S are Byzantine). The execution continues as in $E_{\text{CLEAN-V}}$ until p_t delivers v. Then, processes in S misbehave (they are Byzantine) and do three things: (1) they change their states to what they were at the end of $E_{\text{CLEAN-W}}$ (this is possible because no process in T signed in $E_{\text{CLEAN-W}}$), (2) they write to their registers in shared memory the same last values that they wrote in $E_{\text{CLEAN-W}}$, and (3) they send the same messages they did in $E_{\text{CLEAN-W}}$. Intuitively, processes in S pretend that s broadcast w. Let t be the time at this point; we refer to time t in the next execution. Now, we pause processes in S and let all other processes execute, including D which had been sleeping. Since p_t delivered v and processes in D are correct, they eventually deliver v as well.

[2] If more than two processes crashed initially, pick any two arbitrarily.

Execution E_{BAD}. Processes in $T \cup \{s\}$ are Byzantine and the rest are correct. Initially, the execution is identical to $E_{\text{CLEAN-W}}$, except that (1) processes in D are sleeping not crashed, and (2) processes in $T \cup \{s\}$ do not send messages to processes in D. Execution continues as in $E_{\text{CLEAN-W}}$ until processes in S (which are correct) deliver w. Then, processes in $T \cup \{s\}$ misbehave and do three things: (1) they change their states to what they were in $E_{\text{MIXED-V}}$ at time t – this is possible because in $E_{\text{CLEAN-V}}$ (and therefore in all values and messages they had by time t in $E_{\text{MIXED-V}}$), no non-sender process signed, and in particular, there were no signatures by any process in $S \setminus \{s\}$; (2) they write to the registers in shared memory the same values that they have in $E_{\text{MIXED-V}}$ at time t; and (3) they send all messages they did in $E_{\text{MIXED-V}}$ up to time t. Intuitively, processes in $T \cup \{s\}$ pretend that s broadcast v. Now, processes in D start executing. In fact, execution continues as in $E_{\text{MIXED-V}}$ from time t onward, where processes is S are paused and all other processes execute (including D). Because these processes cannot distinguish the execution from $E_{\text{MIXED-V}}$, eventually they deliver v. Note that processes in D are correct and they deliver v, while processes in S are also correct and deliver w – contradiction. ◀

In the final stage of the proof, we leverage Lemma 4.2 to construct an execution in which many processes sign. This is done by allowing some process to be poised to sign, and then pausing it and letting a new process start executing. Thus, we apply Lemma 4.2 $f-1$ times to incrementally build an execution in which $f-1$ correct processes sign.

▶ **Theorem 4.3.** *Any algorithm that solves Reliable Broadcast in the M&M model with $n = 2f+1$, $f \geq 1$ has an execution in which at least $f-1$ correct non-sender processes sign.*

Proof. If $f = 1$, the result is trivial; it requires $f-1=0$ processes to sign.

Now consider the case $f \geq 2$. If A has an execution in which at least $f+1$ correct processes sign, then we are done. Now suppose A has no execution in which at least $f+1$ correct processes sign. Consider the following execution of A.

All processes and s are correct. Initially, s broadcasts v. Then processes $s, p_1 \dots p_f$ participate, and the rest are delayed. This execution is indistinguishable to $s, p_1 \dots p_f$ from one in which the rest of the processes crashed. Therefore, by Lemma 4.2, some process in $p_1 \dots p_f$ eventually signs. Call p_1 the first process that signs. We continue the execution until p_1's next step is to make its signature visible. Then, we pause p_1, and let p_{f+1} begin executing. Again, this execution is indistinguishable to $s, p_2 \dots p_{f+1}$ from one in which the rest of the processes crashed, so by Lemma 4.2, eventually some process in $p_2 \dots p_{f+1}$ creates a signature and makes it visible. We let the first process to do so reach the state in which it is about to make its signature visible, and then pause it, and let p_{f+2} start executing.

We continue in this way, each time pausing p_i as it is about to make its signature visible, and letting p_{f+i} begin executing. We can apply Lemma 4.2 as long as two processes have not participated yet. At that point, $f-1$ processes are poised to make their signatures visible. We then let these $f-1$ processes each take one step. This yields an execution of A in which $f-1$ correct non-sender processes sign. ◀

5 Broadcast Algorithms

In this section we present solutions for Consistent and Reliable Broadcast. We first implement Consistent Broadcast in Section 5.1; then we use it as a building block to implement Reliable Broadcast, in Section 5.2. We prove the correctness of our algorithms in the full version of our paper [4]. For both algorithms, we first describe the general execution outside the common case, which captures behavior in the worst executions; we then describe how delivery happens fast in the common case (without signatures).

Process roles in broadcast. We distinguish between three process roles in our algorithms: sender, receiver, and replicator. This is similar in spirit to the proposer-acceptor-learner model used by Paxos [38]. Any process may play any number of roles; if all processes play all three roles, then this becomes the standard model. The sender calls *broadcast*, the receivers call *deliver*, and the replicators help guarantee the properties of broadcast. By separating replicators (often servers) from senders and receivers (often clients or other servers), we improve the practicality of the algorithms: clients, by not fulfilling the replicator role, need not remain connected to disseminate information from other clients. Unless otherwise specified, n and f refer only to replicators; independently, the sender and any number of receivers can also be Byzantine. Receivers cannot send or write any values, as opposed to the sender and replicators, but they can read the shared memory and receive messages.

Background signatures. Our broadcast algorithms produce signatures in the background. We do so to allow the algorithms to be signature-free in the common case. Indeed, in the common case, receivers can deliver a message without waiting for background signatures. However, outside the common case, these signatures must still be produced by the broadcast algorithms in case some replicators are faulty or delayed. Both algorithms require a number of signatures that matches the bounds in Section 4 within constant factors.

5.1 Consistent Broadcast

We give an algorithm for Consistent Broadcast that issues no signatures in the common case, when there is synchrony and no replicator is faulty. Outside this case, only the sender signs.

Algorithm 1 shows the pseudocode. The broadcast and deliver events are called *cb-broadcast* and *cb-deliver*, to distinguish them from *rb-broadcast* and *rb-deliver* of Reliable Broadcast. Processes communicate by sharing an array of *slots*: process i can write to *slots[i]*, and can read from all slots. To refer to its own slot, a processes uses index *me*. The sender s uses its slot to broadcast its message while replicators use their slot to replicate the message. Every slot has two sub-slots – each a SWMR atomic register – one for a message (*msg*) and one for a signature (*sgn*).

To broadcast a message m, the sender s writes m to its *msg* sub-slot (line 6). Then, in the background, s computes its signature for m and writes it to its *sgn* sub-slot (line 9). The presence of *msg* and *sgn* sub-slots allow the sender to perform the signature computation in the background. Sender s can return from the broadcast while this background task executes.

The role of a correct replicator is to copy the sender's message m and signature σ, provided σ is valid. The copying of m and σ (lines 12–19) are independent events, since a signature may be appended in the background, i.e., later than the message. The fast way to perform a delivery does not require the presence of signatures. Note that correct replicators can have mismatching values only when s is Byzantine and overwrites its memory.

A receiver p scans the slots of the replicators. It delivers message m when the content of a majority $(n-f)$ of replicator slots contains m and a valid signature by s for m, and no slot contains a different message $m', m' \neq m$ with a valid sender signature (line 28). Slots with sender signatures for $m' \neq m$ result in a no-delivery. This scenario indicates that the sender is Byzantine and is trying to equivocate. Slots with signatures not created by s are ignored so that a Byzantine replicator does not obstruct p from delivering.

When there is synchrony and both the sender and replicators follow the protocol, a receiver delivers without using signatures. Specifically, delivery in the fast path occurs when there is unanimity, i.e., all $n = 2f + 1$ replicators replicated value m (line 25), regardless

Algorithm 1 Consistent Broadcast Algorithm with sender s.

```
1   Shared:
2   slots - $n$ array of "slots"; each slot is a 2-tuple (msg, sgn) of SWMR atomic registers, initialized
        ↪ to $(\bot, \bot)$.

4   Sender code:
5   cb-broadcast(m):
6       slots[me].msg.write(m)
7       In the background:
8           $\sigma$ = compute signature for m
9           slots[me].sgn.write($\sigma$)

11  Replicator code:
12  while True:
13      m = slots[s].msg.read()
14      if m $\neq \bot$:
15          slots[me].msg.write(m)
16      sign = slots[s].sgn.read()
17      val = slots[me].msg.read()
18      if val $\neq \bot$ and sign $\neq \bot$ and sign is a valid signature for val:
19          slots[me].sgn.write(sign)

21  Receiver code:
22  while True:
23      others = scan()
24      if others[i].msg has the same value m for all i in $\Pi$: // Fast path
25          cb-deliver(m); break
26      if others contains at least $n - f$ signed copies of the same value m
27          and ($\nexists$i: others[i].sgn is a valid signature for others[i].msg and others[i].msg $\neq$ m):
28          cb-deliver(m); break

30  scan():
31      others = [slots[i].(msg, sgn).read() for i in $\Pi$]
32      done = False
33      while not done:
34          done = True
35          for i in $\Pi$:
36              if others[i] == $\bot$:
37                  others[i] = slots[i].(msg, sgn).read()
38                  if others[i] $\neq \bot$:
39                      done = False
40      return others
```

of whether a signature is provided by s. A correct sender eventually appends σ, and $n - f$ correct replicators eventually copy σ over, allowing another receiver to deliver m via the slow path, even if a replicator misbehaves, e.g., removes or changes its value.

An important detail is the use of a snapshot to read replicators' slots (line 23), as opposed to a simple collect. The scan operation is necessary to ensure that concurrent reads of the replicators' slots do not return views that can cause correct receivers to deliver different messages. To see why, imagine that the scan at line 23 is replaced by a simple collect. Then, an execution is possible in which correct receiver p_1 reads some (correctly signed) message m_1 from $n - f$ slots and finds the remaining slots empty, while another correct receiver p_2 reads $m_2 \neq m_1$ from $n - f$ slots and finds the remaining slots empty. In this execution, p_1 would go on to deliver m_1 and p_2 would go on to deliver m_2, thus breaking the consistency property. We present such an execution in detail in [4].

To prevent scenarios where correct receivers see different values at a majority of replicator slots, the *scan* operation works as follows (lines 30–40): first, it performs a collect of the slots. If all the slots are non-empty, then we are done. Otherwise, we re-collect the *empty slots* until no slot becomes non-empty between two consecutive collects. This suffices to avoid the problematic scenario above and to guarantee liveness despite f Byzantine processes.

5.2 Reliable Broadcast

We now give an algorithm for Reliable Broadcast that issues no signatures in the common case, and issues only $n+1$ signatures in the worst case. Algorithm 3 (part of Appendix B) shows the pseudocode.

Processes communicate by sharing arrays *Echo* and *Ready*, which have the same structure of sub-slots as *slots* in Section 5.1. *Echo[i]* and *Ready[i]* are writable only by replicator i, while the sender s communicates with the replicators using an instance of Consistent Broadcast (CB) and does not access *Echo* or *Ready*. In this CB instance, s invokes *cb-broadcast*, acting as sender for CB, and the replicators invoke *cb-deliver*, acting as receivers for CB.

To broadcast a message, s *cb-broadcast*s $\langle \textsc{Init}, m\rangle$ (line 6). Upon delivering the sender's message $\langle \textsc{Init}, m\rangle$, each replicator writes m to its *Echo msg* sub-slot (line 13). Then, in the background, a replicator computes its signature for m and writes it to its *Echo sgn* sub-slot (line 16). By the consistency property of Consistent Broadcast, if two correct replicators r and r' deliver $\langle \textsc{Init}, m\rangle$ and $\langle \textsc{Init}, m'\rangle$ respectively, from s, then $m = m'$. Essentially, correct replicators have the same value or $\perp$ in their *Echo msg* sub-slot.

Next, replicators populate their *Ready* slots with a *ReadySet*. A replicator r constructs such a *ReadySet* from the $n-f$ signed copies of m read from the *Echo* slots (lines 19–28). In the background, r reads the *Ready* slots of other replicators and copies over – if r has not written one already – any valid *ReadySet* (line 36). Thus, totality is ensured (Definition 3.2), as the *ReadySet* created by any correct replicator is visible to all correct receivers.

To deliver m, a receiver p reads $n-f$ valid *ReadySet*s for m (line 45).[3] This is necessary to allow a future receiver p' deliver a message as well. Suppose that p delivers m by reading a single valid *ReadySet* R.[4] Then, the following scenario prevents p' from delivering: let sender s be Byzantine and let R be written by a Byzantine replicator r. Moreover, let a *single* correct replicator have *cb-deliver*ed m, while the remaining correct replicators do not deliver at all, which is allowed by the properties of Consistent Broadcast. So, the *ReadySet* contains values from a single correct replicator and f other Byzantine replicators. If r removes R from its *Ready* slot, it will block the delivery for p' since no valid *ReadySet* exists in memory.

A receiver p can also deliver the sender's message m using a fast path. The signature-less fast path occurs when p reads m from the *Echo* slots of all replicators (line 43), and the delivery of the $\textsc{Init}$ message by the replicators is done via the fast path of Consistent Broadcast. This is the common case, when replicators are not faulty and replicate messages timely. Note that p delivering m via the fast path does not prevent another receiver p' from delivering. Process p' delivers m via the fast path if all the *Echo* slots are in the same state as for p. Otherwise, e.g., some Byzantine replicators overwrite their *Echo* slots, p' delivers m by relying on the $n-f$ correct replicators following the protocol (line 45).

6 Consensus

We now give an algorithm for consensus using Consistent Broadcast as its communication primitive, rather than the commonly used primitive, Reliable Broadcast. Our algorithm is based on the PBFT algorithm [20, 21] and proceeds in a sequence of (consecutive) views. It has four features: (1) it works for $n = 2f+1$ processes, (2) it issues no signatures in the common case, (3) it issues $O(n^2)$ signatures on a view-change and (4) it issues $O(n)$ required background signatures within a view.

[3] In contrast to Algorithm 1, receivers need not use the *scan* operation when gathering information from the replicators' *Ready* slots because there can only be a single value with a valid *ReadySet*.

[4] A similar argument that breaks totality applies if p were to deliver m by reading $n-f$ signed values of m in the replicators' *Echo* slots.

Our algorithm uses a sequence of Consistent Broadcast instances indexed by a broadcast sequence number k. When process p broadcasts its k^{th} message m, we say that p broadcasts (k, m). We assume the following ordering across instances, which can be trivially guaranteed: (**FIFO delivery**) For $k \geq 1$, no correct process delivers (k, m_k) from p unless it has delivered (i, m_i) from p, for all $i < k$.

Algorithm 2 shows the pseudocode. The full version of our paper [4] has its correctness proof. The protocol proceeds in a sequence of consecutive *views*. Each view has a primary process, defined as the view number mod n (line 6). A view has two phases, Prepare and Commit. There is also a view-change procedure initiated by a ViewChange message.

When a process is the primary (line 9), it broadcasts a Prepare message with its estimate *init* (line 11), which is either its input value or a value acquired in the previous view (line 10). Upon receiving a valid Prepare message, a replica broadcasts a Commit message (line 20) with the estimate it received in the Prepare message. We define a Prepare to be valid when it originates from the primary and either (a) $view = 0$ (any estimate works), or (b) $view > 0$ and the estimate in the Prepare message has a proof from the previous view. The extended paper [4] details the conditions for a message to be valid. When a replica receives an invalid Prepare message from the primary or times out, it broadcasts a Commit message with $\perp$. If a replica accepts a Prepare message with *val* as estimate and $n - f$ matching Commit messages (line 24), it decides on *val*.

Algorithm 2 Consensus protocol based on Consistent Broadcast ($n = 2f + 1$)

```
1   propose(v_i):
2       view_i = 0; est_i = ⊥; aux_i = ⊥
3       proof_i = ∅; vc_i= (0, ⊥, ∅)
4       decided_i = False
5       while True:
6           p_i = view_i % n

8           // Phase 1
9           if p_i == i:
10              init_i = est_i if est_i ≠ ⊥ else v_i
11              cb-broadcast(⟨Prepare, view_i, init_i, proof_i⟩)
12          wait until receive valid ⟨Prepare, view_i, val, proof⟩ from p_i or timeout on p_i
13          if received valid ⟨Prepare, view_i, val, proof⟩ from p_i:
14              aux_i = val
15              vc_i = (view_i,val,proof)
16          else:
17              aux_i = ⊥

19          // Phase 2
20          cb-broadcast(⟨Commit, view_i, aux_i⟩)
21          wait until receive valid ⟨Commit, view_i, *⟩ from $n - f$ processes
22                  and (∀j: receive valid ⟨Commit, view_i, *⟩ from j or timeout on j)
23          ∀j: R_i[j] = val if received valid ⟨Commit, view_i, val⟩ from j else ⊥
24          if ∃val ≠ ⊥ : $\#_{val}$(R_i)≥ $n - f$ and aux_i == val:
25              try_decide(val)

27          // Phase 3
28          cb-broadcast(⟨ViewChange, view_i + 1, vc_i⟩$_{\sigma_i}$)
29          wait until receive $n - f$ non-conflicting view-change certificates for view_i + 1
30          proof_i = set of non-conflicting view-change certificates
31          est_i = val in proof_i associated with the highest view
32          view_i = view_i + 1

34      In the background:
35          when cb-deliver valid ⟨ViewChange, view', vc⟩$_{\sigma_j}$ from j:
36              cb-broadcast(⟨ViewChangeAck, d⟩$_{\sigma_i}$)  // d is the view-change message being ACKed

38  try_decide(val):
39      if not decided_i:
40          decided_i = True
41          decide(val)
```

The view-change procedure ensures that all correct replicas eventually reach a view with a correct primary and decide. It uses an acknowledgement phase similar to PBFT with MACs [21]. While in [21] the mechanism is used so that the primary can prove the authenticity of a view-change message sent by a faulty replica, we use this scheme to ensure that (a) a faulty participant cannot lie about a committed value in its ViewChange message and (b) valid ViewChange messages can be received by all correct replicas.

A replica starts a view-change by broadcasting a signed ViewChange message with its view-change tuple (line 28). The view-change tuple *(view, val, proof$_{val}$)* is updated when a replica receives a valid Prepare message (line 15). It represents the last non-empty value a replica accepted as a valid estimate and the view when this occurred. We use the value's proof, *proof$_{val}$*, to prevent a Byzantine replica from lying about its value: suppose a correct replica decides *val* in view v, but in view $v + 1$, the primary p is silent, and so no correct replica hears from p; without the proof, a Byzantine replica could claim to have accepted *val′* in $v + 1$ from p during the view-change to $v + 1$, thus overriding the decided value *val*.

When a replica receives a valid ViewChange message, it responds by broadcasting a signed ViewChangeAck containing the ViewChange message (line 36). A common practice is to send a digest of this message instead of the entire message [20]. We define a ViewChange message m from p to be valid when the estimate in the view-change tuple corresponds to the value broadcast by p in its latest non-empty Commit and m's proof is valid. We point out that, as an optimization, this proof can be removed from the view-change tuple and be provided upon request when required to validate ViewChange messages. For instance, in the scenario described above, when a (correct) replica r did not accept *val′* in view $v + 1$, as claimed by the Byzantine replica r', r can request r' to provide a proof for *val′*.

A view-change certificate consists of a ViewChange message and $n - f - 1$ corresponding ViewChangeAck messages. This way, each view-change certificate has the contribution of at least one correct replica, who either produces the ViewChange message or validates a ViewChange message. Thus, when a correct replica r receives a view-change certificate relayed by the primary, r can trust the contents of the certificate.

To move to the next view, a replica must gather a set of $n - f$ non-conflicting view-change certificates Ψ. This step is performed by the primary of the next view, who then includes this set with its Prepare message for the new view. Two view-change certificates conflict if their view-change messages carry a tuple with different estimates ($\neq \bot$), valid proof, and same view number. If the set Ψ consists of tuples with estimates from different views, we select the estimate associated with the highest view. Whenever any correct replica decides on a value *val* within a view, the protocol ensures a set of non-conflicting view-change certificates can be constructed only for *val* and hence the value is carried over to the next view(s).

Discussion

We discuss how Algorithm 2 achieves the four features mentioned at the beginning of Section 6. The first feature (the algorithm solves consensus with $n = 2f + 1$ processes) follows directly from the correctness of the algorithm. The second feature (the algorithm issues no signatures in the common case) holds because in the common case, processes will be able to deliver the required Prepare and Commit messages and decide in the first view, without having to wait for any signatures to be produced or verified. The third feature (the algorithm issues $O(n^2)$ signatures on view-change) holds because, in the worst case, during a view change each process will sign and broadcast a ViewChange message, thus incurring $O(n)$ signatures in total, and, for each such message, each other process will sign and broadcast a ViewChangeAck message, thus incurring $O(n^2)$ signatures. The fourth feature states that the algorithm issues $O(n)$ required background signatures within a view. These signatures are incurred by *cb-broadcast*ing PREPARE and COMMIT messages. In every view, correct processes broadcast a COMMIT message, thus incurring $n - f = O(n)$ signatures in total.

To the best of our knowledge, no existing algorithm has achieved all these four features simultaneously. The only broadcast-based algorithm which solves consensus with $n = 2f + 1$ processes that we are aware of, that of Correia et al. [27], requires $O(n)$ calls to Reliable Broadcast before any process can decide; this would incur $O(n^2)$ required background signatures when using our Reliable Broadcast implementation – significantly more than our algorithm's $O(n)$ required background signatures.

At this point, the attentive reader might have noticed that our consensus algorithm uses some techniques that bear resemblance to our Reliable Broadcast algorithm in Section 5. Namely, the primary of a view *cb-broadcast*s a PREPARE message which is then echoed by the replicas in the form of COMMIT messages. Also, during view change, a replica's VIEWCHANGE message is echoed by other replicas in the form of VIEWCHANGEACK messages. This is reminiscent of the Init-Echo technique used by our Reliable Broadcast algorithm.

Thus, the following question arises: Can we replace each instance of the witnessing technique in our algorithm by a single Reliable Broadcast call and thus obtain a conceptually simpler algorithm, which also satisfies the three above-mentioned properties? Perhaps surprisingly, the resulting algorithm is incorrect. It allows an execution which breaks agreement in the following way: a correct replica p_1 *rb-delivers* some value v from the primary and decides v; sufficiently many other replicas time out waiting for the primary's value and change views without "knowing about" v; in the next view, the primary *rb-broadcasts* v', which is delivered and decided by some correct replica p_2.

Intuitively, by using a single Reliable Broadcast call instead of multiple Consistent Broadcast calls, some information is not visible to the consensus protocol. Specifically: while it is true that, in order for p_1 to deliver v in the execution above, $n - f$ processes must echo v (and thus they "know about" v), this knowledge is however encapsulated inside the Reliable Broadcast abstraction and not visible to the consensus protocol. Thus, the information cannot be carried over to the view-change, even by correct processes. This intuition provides a strong motivation to use Consistent Broadcast – rather than Reliable Broadcast – as a first-class primitive in the design of Byzantine-resilient agreement algorithms.

7 Conclusion

A common tool to address Byzantine failures is to use signatures or lots of replicas. However, modern hardware makes these techniques prohibitive: signatures are much more costly than network communication, and excessive replicas are expensive. Hence, we seek algorithms that minimize the number of signatures and replicas. We applied this principle to broadcast primitives in the message-and-memory model, and derived algorithms that avoid signatures in the common case, use nearly-optimal number of signatures in the worst case, and require only $n = 2f+1$ replicas. We proved worst-case lower bounds on the number of signatures required by Consistent Broadcast and Reliable Broadcast, showing a separation between these problems. We presented the first Byzantine consensus algorithm for $n = 2f+1$ without signatures in the common case. A novelty of our protocol is the use of Consistent Broadcast instead of Reliable Broadcast, which resulted in fewer signatures than existing consensus protocols based on Reliable Broadcast.

References

1 Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 2018.

2 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, 2019.

3 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 599–616, 2020.

4 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. Frugal Byzantine Computing. *arXiv preprint*, 2021. `arXiv:2108.01330`.

5 Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N Wright. Tight bounds for shared memory systems accessed by Byzantine processes. *Distributed computing (DIST)*, 18(2), 2005.

6 Hagit Attiya, Sweta Kumari, and Noa Schiller. Optimal resilience in systems that mix shared memory and message passing. *arXiv preprint*, 2020. `arXiv:2012.10846`.

7 Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4), 2015.

8 Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 349–359, 2016.

9 Naama Ben-David and Kartik Nayak. Brief announcement: Classifying trusted hardware via unidirectional communication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

10 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, 1983.

11 Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3), 2009.

12 Bitcoin Core Developers. Optimized C library for ECDSA signatures and secret/public key operations on curve secp256k1. `https://github.com/bitcoin-core/secp256k1`.

13 Zohir Bouzid, Damien Imbs, and Michel Raynal. A necessary condition for Byzantine k-set agreement. *Information Processing Letters*, 116(12), 2016.

14 Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

15 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2), 1987.

16 Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 12–26, 1983.

17 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4), 1985.

18 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

19 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2001.

20 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1999.

21 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

22 Certicom Research. Standards for efficient cryptography. https://www.secg.org/sec2-v2.pdf, 2010.

23 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), 1996.

24 Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, 2008. URL: http://www.usenix.org/events/usenix08/tech/full_papers/chun/chun.pdf.

25 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, 2007. doi:10.1145/1294261.1294280.

26 Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*, 2004. doi:10.1109/RELDIS.2004.1353018.

27 Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous Byzantine consensus with $2f + 1$ processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

28 Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

29 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

30 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

31 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), 1988.

32 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985.

33 Vassos Hadzilacos, Xing Hu, and Sam Toueg. Optimal Register Construction in M&M Systems. In *International Conference on Principles of Distributed Systems (OPODIS)*, volume 153, pages 28:1–28:16, 2020.

34 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, 2012. doi:10.1145/2168836.2168866.

35 Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2), 2001.

36 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

37 Leslie Lamport. The weak Byzantine generals problem. *Journal of the ACM (JACM)*, 30(3), 1983.

38 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

39 Leslie Lamport. Fast Paxos. *Distributed computing (DIST)*, 19(2), 2006.

40 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

41 linux-rdma. Rdma benchmarking utility. https://github.com/linux-rdma/perftest.

42 Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

43 Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. Objects shared by Byzantine processes. *Distributed computing (DIST)*, 16(1), 2003.

44 Mellanox. Network benchmarking utility. `https://github.com/Mellanox/sockperf`.

45 Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 2–9, 2014.

46 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2), 1980.

47 Michel Raynal and Jiannong Cao. One for all and all for one: Scalable consensus in a hybrid communication model. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 464–471. IEEE, 2019.

48 Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68—-80, 1994. `doi:10.1145/191177.191194`.

49 Blockchain hardware accelerator. `https://www.xilinx.com/products/intellectual-property/1-175rk99.html`. Accessed 2021-02-15.

50 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed computing (DIST)*, 2(2):80–94, 1987.

51 Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 49(8):357–368, 2014.

52 Sam Toueg. Randomized Byzantine agreements. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–178, 1984.

53 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1), 2013. `doi:10.1109/TC.2011.221`.

A APPENDIX: Latency

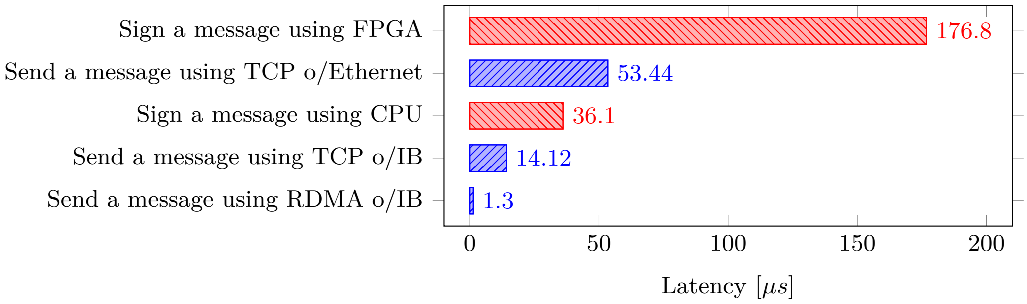


Figure 1 RDMA communication is significantly faster than signature creation using CPU or hardware acceleration (FPGA). The graph shows the latency of sending or signing a 32-byte message. IB means Infiniband, a faster interconnect than Ethernet found in data centers. TCP latencies are obtained using sockperf [44]. RDMA latency is obtained using perftest [41]. Signatures use optimized implementations for CPU [12] and FPGA [49] of the ECDSA algorithm on the secp256k1 elliptic curve [22]. An FPGA improves the throughput of signature creation (not shown in figure), but not its latency, due to their relatively low clock speeds (compared to CPUs) and the non-parallelizable nature of algorithms for digital signature.

B APPENDIX: Reliable Broadcast Algorithm

Algorithm 3 Reliable Broadcast Algorithm with sender s.

```
Shared:
Echo, Ready - $n$ array of "slots"; each slot is a 2-tuple (msg, sgn) of SWMR atomic registers,
      ↪ initialized to $(\bot, \bot)$.

Sender code:
rb-broadcast(m):
    cb-broadcast(⟨INIT,m⟩)

Replicator code:
state = WaitForSender // ∈{WaitForSender,WaitForEchos}
while True:
    if state == WaitForSender:
        if cb-delivered ⟨INIT,m⟩ from s:
            Echo[me].msg.write(m)
            In the background:
                $\sigma =$ compute signature for m
                Echo[me].sgn.write($\sigma$)
            state = WaitForEchos

    if state == WaitForEchos:
        ReadySet = $\emptyset$
        for i $\in \Pi$:
            other = Echo[i].(msg,sgn).read()
            if other.msg == m and other.sgn is m validly signed by i:
                ReadySet.add((i,other))

        if size(ReadySet) $\geq n - f$:
            ready = True
            Ready[me].msg.write(ReadySet)

In the background:
    while True
        if not ready:
            others = [Ready[i].msg.read() for i in $\Pi$]
            if $\exists$i: others[i] is a valid ReadySet:
                ready = True
                Ready[me].msg.write(others[i])

Receiver code:
while True:
    others = [Echo[i].msg.read() for i in $\Pi$]
    proofs = [Ready[i].msg.read() for i in $\Pi$]
    if others contains $n$ matching values m: // Fast path
        rb-deliver(m); break
    if proofs contains $n - f$ valid ReadySet for the same value m:
        rb-deliver(m); break
```

Lower Bounds for Shared-Memory Leader Election Under Bounded Write Contention

Dan Alistarh ✉
IST Austria, Klosterneuburg, Austria

Rati Gelashvili ✉
Novi Research, Menlo Park, CA, USA

Giorgi Nadiradze ✉
IST Austria, Klosterneuburg, Austria

Abstract

This paper gives tight logarithmic lower bounds on the solo step complexity of leader election in an asynchronous shared-memory model with single-writer multi-reader (SWMR) registers, for both deterministic and randomized obstruction-free algorithms. The approach extends to lower bounds for deterministic and randomized obstruction-free algorithms using multi-writer registers under bounded write concurrency, showing a trade-off between the solo step complexity of a leader election algorithm, and the worst-case number of stalls incurred by a processor in an execution.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Lower Bounds, Leader Election, Shared-Memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.4

Funding *Dan Alistarh*: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).
Giorgi Nadiradze: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).

Acknowledgements The authors would like to thank the DISC anonymous reviewers for their useful feedback and comments.

1 Introduction

Leader election is a classic distributed coordination problem, in which a set of n processors must cooperate to decide on the choice of a single "leader" processor. Each processor must output either a *win* or *lose* decision, with the property that, in any execution, a single processor may return *win*, while all other processors have to return *lose*. Moreover, any processor returns *win* in *solo* executions, in which it does not observe any other processor.

Due to its fundamental nature, the time and space complexity of variants of this problem in the classic asynchronous shared-memory model has been the subject of significant research interest. Leader election and its linearizable variant called *test-and-set* are weaker than consensus, as processors can decide without knowing the leader's identifier. Test-and-set differs from leader election in that no processor may return *lose* before the eventual winner has joined the computation, and has consensus number two. It therefore cannot be implemented deterministically wait-free [19]. Tromp and Vitányi gave the first *randomized* algorithm for *two-processor* leader election [29], and Afek, Gafni, Tromp and Vitányi [1] generalized this approach to n processors, using the tournament tree idea of Peterson and Fischer [27].

Their algorithm builds a complete binary tree with n leaves; each processor starts at a leaf, and proceeds to compete in two-processor leader-election objects located at nodes, returning *lose* whenever it loses at such an object. The winner at the root returns *win*. Since each two-processor object can be resolved in expected constant time, their algorithm has expected step complexity $O(\log n)$ against an adaptive adversary. Moreover, their algorithm only uses *single-reader multiple-writer (SWMR)* registers: throughout any execution, any register may only be written by a single processor, although it may be read by any processor.

Follow-up work on time upper bounds has extended these results to the adaptive setting, showing logarithmic expected step complexity in the number of participating processors k [4, 16]. Further, Giakkoupis and Woelfel [16] showed that, if the adversary is oblivious to the randomness used by the algorithm, $O(\log^\star k)$ step complexity is achievable, improving upon a previous sub-logarithmic upper bound by Alistarh and Aspnes [2]. Another related line of work has focused on the *space complexity* of this problem, which is now resolved. Specifically, it is known that $\Omega(\log n)$ distinct registers are *necessary* [28, 16], and a breakthrough result by Giakkoupis, Helmi, Higham, and Woelfel [15] provided the first asymptotically matching upper bound of $O(\log n)$, improving upon an $O(\sqrt{n})$ algorithm by the same authors [14].

The clear gap in the complexity landscape for this problem concerns time complexity lower bounds. Specifically, in the standard case of an adaptive adversary, the best known upper bound is the venerable tournament-tree algorithm we described above [1], which has $O(\log n)$ expected time complexity and uses SWMR registers. It is not known whether one can perform leader election in classic asynchronous shared-memory faster than a tournament.[1] Due to the simplicity of the problem, none of the classic lower bound approaches, e.g. [23, 24, 22], apply, and resolving the time complexity of shared-memory leader election is known to be a challenging open problem [2, 16]. Moreover, given that the step complexities of shared-memory consensus [8] and renaming [3] have been resolved, leader election remains one of the last basic objects for which no tight complexity bounds are known.

We show tight logarithmic lower bounds on the step complexity of leader election in asynchronous shared-memory with SWMR registers. Our motivating result is a natural potential argument showing that any *deterministic* obstruction-free algorithm for leader election – in which processors must return if they execute enough solo steps – must have worst-case step complexity $\Omega(\log n)$ *in solo executions*, that is, even if processors execute in the absence of concurrency, as long as registers are SWMR.

Our main contribution is a new and non-trivial technique showing that a similar statement holds for *randomized* algorithms: in the same model, any *obstruction-free* algorithm for leader election has worst-case expected cost $\Omega(\log n)$. In this case as well, the lower bound holds in terms of expected step complexity *in solo executions.* The lower bound technique is based on characterizing the expected length of solo executions by analyzing the number of reads and writes over distinct registers required by a correct algorithm.

These are the first non-trivial lower bounds on the time complexity of classic shared-memory leader election, although they assume restrictions on the algorithms. They are both matched asymptotically by the tournament-tree approach, as the algorithm of [1] can be modified to be deterministic obstruction-free, by using two-processor obstruction-free leader election objects. This essentially shows that the tournament strategy is optimal for SWMR registers. The results will also apply to the case where algorithms may employ stronger two-processor read-modify-write primitives, such as two-processor test-and-set operations instead of reads and exclusive writes. Interestingly, the result holds for a *weak* version of leader election, in which all processors may return *lose* if they are in a *contended* execution.

[1] Sub-logarithmic step complexity is achievable in other models, e.g. distributed and cache-coherent shared-memory [17] or message-passing [5].

The main limitation of the approach concerns the SWMR restriction on the registers used by the algorithm. We investigate relaxations of this, and show that, for deterministic algorithms, if κ is the maximum number of processors which might be poised to write to a register in any given execution, then any algorithm will have worst-case solo step complexity $\Omega((\log n)/\kappa)$. Conversely, assuming κ is super-constant with respect to n, any algorithm with $O((\log n)/\kappa)$ solo step complexity will have an execution in which $\Omega(\kappa)$ distinct processors are poised to write concurrently to the same register. Since this latter quantity is an asymptotic lower bound on the worst-case *stall complexity* at a processor [13],[2] this yields a logarithmic trade-off between the worst-case slow-down due to steps in a solo execution, and the worst-case slow-down at a processor due to high register contention, measured in stalls, for any deterministic algorithm.

We generalize this argument to the *randomized* case as well, to show that, for $\kappa \geq 2$, any algorithm ensuring that at most $\kappa - 1$ worst-case stalls at a processor must have expected step complexity $\Omega((\log n)/\kappa^2)$. In practical terms, our results show that any gain made due to decreased steps on the solo fast-path is paid for by an increase in the worst-case stall complexity at a processor incurred by any obstruction-free leader election algorithm.

Additional Related Work. The previous section already covered known time and space complexity results for the classic leader election problem in the standard asynchronous shared-memory model. This fundamental problem has also been considered in under related models and complexity metrics. Specifically, Golab, Hendler and Woelfel [17] have shown that leader election can be solved using *constant* remote memory references (RMRs) in the cache-coherent (CC) and distributed shared-memory (DSM) models. Their result circumvents our lower bounds due to differences in the model and in the complexity metrics. In the same model, Eghbali and Woelfel [12] have shown that *abortable* leader election requires $\Omega(\log n/\log\log n)$ time in the worst case. The abortability constraint imposes stronger semantics, and they consider a different notion of complexity cost, but multi-writer registers.

In addition, our results are also related to early work by Anderson and Yang [30], who assume bounds on the write contention at each register, and prove $\Omega(\log n)$ lower bounds for a weak version of mutual exclusion, assuming bounded write contention per register. Upon careful consideration, one can obtain that their approach can be used to prove a similar logarithmic lower bound for *obstruction-free leader election* in the read-write model with contention constraints. However, we emphasize that their argument works only for *deterministic algorithms.*

Specifically, relative to this paper, our contribution is the randomized lower bound. The argument of Anderson and Yang [30] does not generalize to randomized algorithms, for the same reason that the simple deterministic argument we provide as motivation does not generalize in the randomized case. Even focusing on the deterministic case, our approach is slightly different than the one of Anderson and Kim: we use covering plus a potential argument, while they use a different covering argument based on eliminating contending processors by leveraging Turan's theorem. However, their approach can provide a better dependency on contention in the bound: $\Omega(\log n/\log\kappa)$ versus $\Omega(\log n/\kappa)$ in our case.

We note that similar trade-offs between contention and step complexity have been studied by Dwork, Herlihy and Waarts [11], and by Hendler and Shavit [18], although in the context of different objects, and for slightly different notions of cost. We believe this paper is the first to approach such questions for randomized algorithms, and for leader election.

[2] If $\kappa \geq 2$ processors are poised to write concurrently to a register, then the last processor to write will incur $\kappa - 1$ stalls.

From the technical perspective, the simple deterministic argument we propose can be viewed as a *covering argument* [23, 10, 9, 24, 7], customized for the leader-election problem, and leveraging the SWMR property. The new observation is the potential argument showing that some processor must incur $\Omega(\log n)$ distinct steps in a solo execution. To our knowledge, the lower bound approach for randomized algorithms is new. The generalized argument for bounded concurrent-write contention implies bounds in terms of the *stall metric* of Ellen, Hendler and Shavit [13], which has also been employed by other work on lower bounds, e.g. [7]. These prior approaches do not apply to leader election.

2 Model, Preliminaries, and Problem Statement

We assume the asynchronous shared-memory model, in which n processors may participate in an execution, $t < n$ of which may fail by crashing. Processors are equipped with unique identifiers, which they may use during the computation. For simplicity, we will directly use the corresponding indices, e.g. i, j, to identify processors in the following, and denote the set of all processors by $\mathcal{P}$. Unless otherwise stated, we assume that processors communicate via atomic read and write operations applied to a finite set of registers. The scheduling of processor steps is controlled by a strong (adaptive) adversary, which can observe the structure of the algorithm and the full state of processors, including their random coin flips, before deciding on the scheduling.

As stated, our approach assumes that the number of processors which may be poised to write to any register during the execution is deterministically bounded. Specifically, for an integer parameter $\kappa \geq 1$, we assume algorithms ensure *κ-concurrent write contention*: in any execution of the algorithm, at most κ processors may be concurrently poised to write to any given register. We note that, equivalently, we could assume that the worst-case write-stall complexity of the algorithms is $\kappa - 1$, as having κ processors concurrently poised to write to a given register necessarily implies that the "last" processor scheduled to write incurs $\kappa - 1$ stalls, one for each of the other writes.

Notice that this assumption implies a (possibly random) mapping between each register and the set of processors which write to it in every execution. For $\kappa = 1$, we obtain a variant of the SWMR model, in which a single processor may write to a given register in an execution. Specifically, we emphasize that we allow this mapping between registers and writers to *change* between executions: different processors may write to the same register, but in different executions. This is a generalization of the classic SWMR property, which usually assumes that the processor-to-registers mapping is fixed across all executions.

Without loss of generality, we will assume that algorithms follow a fixed pattern, consisting of repetitions of the following sequence: 1) a shared read operation, possibly followed by local computation, including random coin flips, and 2) a shared write operation, again possibly followed by local computation and coin flips. Note that any algorithm can be re-written following this pattern, without changing its asymptotic step complexity: if necessary, one can insert dummy read and write operations to dedicated NULL registers.

We measure complexity in terms of processor steps: each shared-memory operation is counted as a step. Total step complexity will count the total number of processor steps in an execution, while individual step complexity, which is our focus, is the number of steps that any single processor may perform during any execution.

We now introduce some basics regarding terminology and notation for the analysis, following the approach of Attiya and Ellen [9]. We view the algorithm as specifying the set of possible states for each processor. At any point in time, for any processor, there exists a single *next step* that the processor is poised to take, which may be either a shared-memory read or write step. Following the step, the processor changes state, based on its previous state, the response received from the shared step (e.g., the results of a read), and its local computation or coin flips. *Deterministic* protocols have the property that the processor state following a step is exclusively determined by the previous state and the result of the shared step, e.g. the value read. *Randomized* protocols have the property that the processor has multiple possible next steps, based on the results of local coin flips following the shared-memory step. Each of these possible next steps has a certain non-zero probability. As standard, we assume that the randomness provided to the algorithm is *finite-precision*, and so, the number of possible next steps at each point is *countable.*[3]

A *configuration* C of the algorithm is completely determined by the state of each processor, and by the contents of each register. We assume that initially all registers have some predetermined value, and thus the *initial* configuration is only determined by the input state (or value) of each processor. Two configurations C and C' are said to be *indistinguishable* to processor p if p has the same state in C and C', and all registers have the same contents in both configurations.

A processor p is said to be *poised* to perform step e, which could be a read or a write, in configuration C if e is the next step that p will perform given C. Given a valid configuration C and a valid next step e by p, we denote the configuration after e is performed by p as Ce. An *execution* E is simply a sequence of such valid steps by processors, starting at the initial configuration. Thus, a configuration is *reachable* if there exists an execution E resulting in C. In the following, we will pay particular attention to *solo* processor executions, that is, executions E in which only a single processor p takes steps.

Our progress requirement for algorithms will be *obstruction-freedom* [20], also known as *solo-termination* [23]. Specifically, an algorithm satisfies this condition if, from any reachable configuration C, any processor p must eventually return a decision in every p-solo extension of C, i.e. in every extension $C\alpha_p$ such that α_p only consists of steps by p.

In the following, we will prove lower bounds for the following simplified variant of the leader election problem.

▶ **Definition 1** (Weak Leader Election). *In the Weak Leader Election problem, each participating processor starts with its own identifier as input, and must return either* win *or* lose*. The following must hold:*

1. *(Leader Uniqueness) In any execution, at most a single processor can return* win*.*
2. *(Solo Output) Any processor must return* win *in any execution in which it executes solo.*

We note that this variant does not fully specify return values in contended executions – in particular, under this definition, all processors may technically return *lose* if they are certain that they are not in a solo execution – and does not require linearizability [21], so it is weaker than test-and-set. Our results will apply to this weaker problem variant.

3 Lower Bound for Deterministic Algorithms

As a warm-up result, we provide a simple logarithmic lower bound for the solo step complexity of leader election with SWMR registers. Specifically, the rest of this section is dedicated to proving the following statement:

▶ **Theorem 2.** *Any deterministic leader election protocol in asynchronous shared-memory with SWMR registers has $\Omega(\log n)$ worst-case solo step complexity.*

[3] Our analysis would also work in the absence of this requirement. However, it appears to be standard, and it will simplify the presentation: it will allow us to sum, rather than integrate, over possible executions.

3.1 Adversarial Strategy

We will specify the lower bound algorithmically, as an iterative procedure that the adversary can follow to create a worst-case execution. More precisely, the adversarial strategy will proceed in steps $t \in \{0, 1, \dots, n-1\}$ and will maintain two sets of processors at each step, the *available set* $\mathcal{V}_t$ and the *frozen set* $\mathcal{F}_t$. In addition, we maintain a prefix of the worst-case execution, which we denote by E_t.

Initially, all processors are in initial state, and placed in the pool of *available* processors $\mathcal{V}_0$, while the set of frozen processors $\mathcal{F}_0$ is empty, and the worst-case execution E_0 is empty as well. In addition, we will associate a *blame counter* $\beta_t(i)$ to each available processor i, initially 0. Intuitively, this represents the number of processors that were placed in the frozen set because of i.

In each step $t \geq 0$, we first identify the processor j whose blame count $\beta_t(j)$ is *minimal* among processors from the available set $\mathcal{V}_t$, breaking ties arbitrarily. We then execute the sequence of solo steps α_j of processor j, until we first encounter a *write step* w_j of j to some register r_j which is *read by some available processor* $k \in \mathcal{V}_t$ *in its solo execution.* Note that the step w_j itself is *not* added to the execution prefix E_t. Below, in Lemma 4, we will show that such a write step by j must necessarily exist: otherwise, we could run j until it returns *win*, without this fact being visible to any other processor in the available set.

Having identified this first write step w_j by j, we "freeze" processor j exactly before w_j, and place it in the frozen set at the next step, $\mathcal{F}_{t+1}$, removing it from $\mathcal{V}_{t+1}$. We then update the worst-case execution prefix to $E_{t+1} = E_t \alpha_j$. Finally, we increment the blame count by 1 for every processor $k \in \mathcal{V}_{t+1}$ with the property that k reads from r_j in its solo execution. At this point, step t is complete, and we can move on to step $t+1$. The process stops when there are no more available processors.

3.2 Analysis

We begin by noting the following invariants, maintained by the adversarial strategy:

▶ **Lemma 3.** *At the beginning of each step t, the adversarial strategy enforces the following:*
1. *All available processors $i \in \mathcal{V}_t$ are in their initial state;*
2. *The contents of all registers read by processors in $\mathcal{V}_t$ during their respective solo executions are the same as in the initial configuration.*

Proof. Both claims follow by the structure of the construction. The first claim follows since the only processor which executes in any step $t \geq 0$ is eliminated from $\mathcal{V}_t$. The second claim follows since, at every step $t \geq 0$, we freeze the corresponding processor j *before* it writes to any register read by any of the remaining processors in $\mathcal{V}_{t+1}$. ◀

Notice that this result practically ensures that the execution prefix generated up to every step t is indistinguishable from the initial configuration for processors in the available set $\mathcal{V}_t$. Next, we show that the strategy is well-defined, in the sense that the step processor w_j specified above must exist at each iteration of the strategy.

▶ **Lemma 4.** *Fix a step t and let j be the chosen processor of minimal blame count $\beta_t(j)$. Then there must exist a step w_j in the solo execution of j which writes to some register r_j which is read by some available processor $k \in \mathcal{V}_t$.*

Proof. We will begin by proving a slightly stronger statement, that is, for *any* processor $k \in \mathcal{V}_t$, there must exist a register r_j^k which is written by j in its solo execution and read by k in its solo execution. We will then choose r_j to be the *first* such register written to by j in its solo execution, and w_j to be the corresponding write step.

Assume for contradiction that there exists a processor $k \in \mathcal{V}_t$, $k \neq j$ which does not read from any registers written to by j in its solo execution. By Lemma 3, the current execution is indistinguishable from a solo execution for j. Thus, if j runs solo from the prefix E_t until completion, j must return *win*. However, if k runs solo after j returns, k also must return *win*, since it does not read from any register which j wrote to, and therefore, by Lemma 3, it observes a solo execution as well. This contradicts the *leader uniqueness* property in the resulting execution.

We have therefore established that *every* other processor $k \in \mathcal{V}_t$ must eventually read from a register r_j^k written to by j in its solo execution. (Notice that these registers need not be distinct with respect to processors.) Let w_j^k be the step where j first writes to r_j^k during its solo execution. To ensure the requirements of the adversarial strategy, it suffices to pick w_j to be the *first* such step w_j^k, in temporal order, in j's solo execution. ◀

We now return to the proof, and focus on the blame counts of available processors at any fixed step $t \geq 0$, $(\beta_t(i))_{i \in \mathcal{V}_t}$. Define the potential Γ_t to be $\sum_{i \in \mathcal{V}_t} 2^{\beta_t(i)}$ at time t.

Since $i \in \mathcal{V}_t$ and $\beta_0(i) = 0$, for all processors i, we have that $\Gamma_0 = n$. Next, we show that, due to the way in which we choose the next processor to be executed, we can always lower bound this potential by n.

▶ **Lemma 5.** *For any step $t \geq 0$, we have $\Gamma_t \geq n$.*

Proof. We will proceed by induction. The base step is outlined above. Fix therefore a step $0 \leq t < n - 1$ such that $\Gamma_t \geq n$.

Again, let $j \in \mathcal{V}_t$ be the processor we freeze at step t. For each $i \in \mathcal{V}_t \setminus \{j\}$, let g_i be the weight by which we incremented the blame count of processor i in this step. By Lemma 4. we have that there exist $i \in \mathcal{V}_t \setminus \{j\}$ such that $g_i = 1$. Further, since we chose to execute the processor j with minimal blame count, we have that $\beta_t(j) \leq \beta_t(i)$. Let us now analyze the difference

$$\Gamma_{t+1} - \Gamma_t = 2^{\beta_t(i)+g_i} - 2^{\beta_t(i)} - 2^{\beta_t(j)} = 2^{\beta_t(i)} - 2^{\beta_t(j)} \geq 0.$$

Hence, $\Gamma_{t+1} \geq \Gamma_t \geq n$, as required. ◀

To complete the proof of Theorem 2, let ℓ be the last remaining non-frozen processor before the process completes, i.e. $\mathcal{V}_{n-1} = \{\ell\}$. By Lemma 5, we have that $\Gamma_{n-1} = 2^{\beta_{n-1}(\ell)} \geq n$, which implies that $\beta_{n-1}(\ell) \geq \log_2 n$. Further, notice that processor ℓ must have performed at least $\beta_{n-1}(\ell)$ *distinct* read operations: for every increment of $\beta_{n-1}(\ell)$, there must exist a unique processor i which wrote to some register r_t^i from which ℓ reads in its solo execution. Since we are assuming SWMR registers, the reads performed by ℓ must be also *unique*. Hence, processor ℓ performs $\log_2 n$ steps in a solo execution, implying an $\Omega(\log n)$ solo step complexity lower bound for the algorithm.
This strongly suggests that the tournament-tree approach is optimal for SWMR registers.

3.3 Discussion

Bounded Concurrent-Write Contention and Stalls. Second, it is interesting to observe what happens to the above argument in the case of *multi-writer* registers. Let $\kappa \geq 1$ be the bound on the concurrent-write contention over any single register, in any execution, that is, on the maximum number of processors which may be concurrently poised to write to a register. Notice that the overall construction and the blaming mechanism would still work. Therefore, the potential lower bound still holds, but in the proof of the last step, the different steps taken by the last processor ℓ do not necessarily need to be distinct. Specifically, we note that a single read step by ℓ may be counted at most κ times, once for each different processor which may be frozen upon its write to the corresponding register. The lower bound is therefore weakened linearly in κ.

▶ **Corollary 6.** *Any deterministic leader election protocol in asynchronous shared-memory where at most $\kappa \geq 1$ may be poised to write to a register concurrently has worst-case solo step complexity $(\log n)/\kappa$. Moreover, if the lower bound construction above implies $(\log n)/\kappa$ worst-case step complexity for a processor, then there must exist an execution in which the concurrent-write contention on some register is κ.*

Recall that, when interpreted in the stall model of [13], having $\kappa \geq 2$ processors poised to write to a register at the same time implies *(write-)stall complexity* $\kappa - 1$ for one of the processors. Thus, this last result implies a logarithmic multiplicative trade-off between the worst-case step complexity of a protocol and its worst-case stall complexity.

Stronger Primitives. Third, we note that this approach can also be extended to deterministic algorithms employing SWMR registers supporting *read*, and *write*, and additionally 2-processor *test-and-set* objects. We can then apply the same freezing strategy, and note that an access to a *test-and-set* object can only lead to freezing a processor and incrementing the blame counter of another processor just once (otherwise there is a combined execution with more than 2 processors accessing it). Hence, we still obtain a lower bound of $\log n$ solo step complexity, i.e. that the tournament tree is the optimal strategy.

4 Lower Bound for Randomized Algorithms

We now shift gears and present our main result, which is a logarithmic expected-time lower bound for randomized obstruction-free algorithms. Our approach in this case will be different, as we are unable to build an explicit worst-case adversarial strategy. Instead, we will argue about the expected length of executions by bounding the expected number of reads over distinct registers required for algorithms to be correct. In turn, this will require a careful analysis of the probability distribution over solo executions of a specific well-chosen structure.

We first focus on the SWMR case, and cover it exclusively in Sections 4.1 to 4.3. We will then provide a generalization to MWMR registers under bounded concurrency in Section 4.4.

4.1 Preliminaries

Fix a processor $p \in \mathcal{P}$. For each p, we define the set $S(p)$ as the set of all possible solo executions of p, and will focus on understanding the probability distribution over reads and writes for executions in $S(p)$. By the *solo output* property of the algorithm (Definition 1), all these executions have to be finite in length. For any possible solo execution e of processor $p \in \mathcal{P}$, $\Pr[e]$ will be used to denote the probability that if we let run p run solo, it will execute e and return. In particular, $\sum_{e \in S(p)} Pr[e] = 1$.

Let $\mathcal{R}$ denote the set of all registers which could be used by the algorithm over all *solo* executions by some processor. Since the randomness provided to the algorithm is finite-precision, the number of possible next steps in every configuration is countable and by the spiral argument, $\mathcal{R}$ must be countable as well[4]. Fix a register $r \in \mathcal{R}$; by definition, r is read or written by some processor during some solo execution. Let $\mathcal{A}(r)$ be a set of all solo executions which read from a register r:

[4] Our argument works even when R isn't countable, but this simplifies notation, e.g. discrete sums.

$$\mathcal{A}(r) = \Big\{e \big| \exists p \in \mathcal{P}[e \in S(p) \wedge \mathrm{read}(r) \in e]\Big\}.$$

We define the *read potential* $\rho(r)$ to roughly count the sum of probabilities that a register is read from during solo executions by any processor. Formally,

$$\rho(r) = \sum_{e \in \mathcal{A}(r)} \Pr[e] = \sum_{p \in \mathcal{P}} \sum_{e_p \in S(p) \cap \mathcal{A}(r)} \Pr[e_p].$$

Analogously, let $\mathcal{B}(r)$ be as a set of all solo executions which write to a register r:

$$\mathcal{B}(r) = \Big\{e \big| \exists p \in \mathcal{P}[e \in S(p) \wedge \mathrm{write}(r) \in e]\Big\}.$$

We define the *write potential* of a register r as

$$\gamma(r) = \sum_{e \in \mathcal{B}(r)} \Pr[e] = \sum_{p \in \mathcal{P}} \sum_{e_p \in S(p) \cap \mathcal{B}(r)} \Pr[e_p].$$

For the simplicity we assume that for any $r \in \mathcal{R}$, $\gamma(r) > 0$ (or alternatively $\mathcal{B}(r) \neq \emptyset$). Otherwise, the reads from r do not change the outcome of the solo executions, and we can assume that they do not use r.

Further, for any given solo execution $e \in S(p)$ of processor p, we define the *trace* of e, $\mathcal{TR}(e)$, as the sequence of registers written by p during e, in the order in which they were written, but omitting duplicate registers. For instance, if in execution e processor p wrote to u_1, then u_2 followed by u_1 again and finally u_3, the trace would be u_1, u_2, u_3 (notice that registers are sorted by the order they are written to for the first time in e). Also, for each register $r \in \mathcal{R}$ and solo execution $e \in \mathcal{B}(r)$, let $\xi_r(e)$ be the index of register r in the trace of e. That is, if $\mathcal{TR}(e) = u_1^e, u_2^e, \ldots, u^e_{|\mathcal{TR}(e)|}$, then $r = u^e_{\xi_r(e)}$.

Our lower bound relies heavily on double-counting techniques. To familiarize the reader with notation and provide some intuition, we isolate and prove the following simple properties of traces. We fix an execution e, and the corresponding notation, as defined above.

▶ **Lemma 7.** *Given the above notation, we have that* $\sum_{r \in \mathcal{TR}(e)} \rho(r) \geq n - 1$.

Proof. Fix a processor $p \in \mathcal{P}$. Recall that every processor $q \in P \setminus \{p\}$ has to read from some register which p writes to in its solo execution e. Otherwise, there is an interleaving of p's solo execution e, followed by q's execution, which neither p nor q can distinguish from their respective solo executions. Therefore, in this interleaved execution, p and q will both return *win*, which leads to a contradiction.

This means that, for every solo execution $e_q \in S(q)$, there exists a register $r \in \mathcal{TR}(e)$, such that *read*$(r) \in e_q$ and hence:

$$\sum_{r \in \mathcal{TR}(e)} \rho(r) \geq \sum_{r \in \mathcal{TR}(e)} \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q) \cap \mathcal{A}(r)} \Pr[e_q] \geq \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q)} \Pr[e_q] = n - 1. \qquad \blacktriangleleft$$

Before proving the next lemma, we provide an intuition from the deterministic setting. For each processor p, assume that there exists $e_p \in S(p)$ such that $\Pr[e_p] = 1$ and consider the sum $\sum_{p \in \mathcal{P}} \sum_{r \in \mathcal{TR}(e_p)} \frac{\rho(r)}{\gamma(r)}$. For each register r, we know that $\frac{\rho(r)}{\gamma(r)}$ appears $\Big|\{p | r \in \mathcal{TR}(e_p)\}\Big| = \Big|\{p | \mathrm{write}(r) \in e_p\}\Big| = \gamma(r)$ times in this summation. Hence, $\sum_{p \in \mathcal{P}} \sum_{r \in \mathcal{TR}(e_p)} \frac{\rho(r)}{\gamma(r)} = \sum_{r \in \mathcal{R}} \frac{\rho(r)}{\gamma(r)} \gamma(r) = \sum_{r \in \mathcal{R}} \rho(r)$.

▶ **Lemma 8.**

$$\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\sum_{r\in\mathcal{TR}(e)}\frac{\rho(r)}{\gamma(r)}=\rho(r). \tag{1}$$

Proof. We have that

$$\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\sum_{r\in\mathcal{TR}(e)}\frac{\rho(r)}{\gamma(r)}=\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\mathbb{1}_{r\in\mathcal{TR}(e)}$$
$$=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\mathbb{1}_{r\in\mathcal{TR}(e)}=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\sum_{p\in\mathcal{P}}\sum_{e\in S(p)}\Pr[e]\mathbb{1}_{\text{write}(r)\in e}$$
$$=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\sum_{e\in\mathcal{B}(r)}\Pr[e]=\sum_{r\in\mathcal{R}}\frac{\rho(r)}{\gamma(r)}\gamma(r)=\sum_{r\in\mathcal{R}}\rho(r).$$

Where in the second equality we simply rearranged the terms. ◀

Finally, we will need the following useful property.

▶ **Lemma 9.** *For any sequence of positive real numbers $x_1, x_2, ...x_m$, we have that*

$$\sum_{i=1}^{m}\frac{x_i}{1+\sum_{j=1}^{i-1}x_j}\geq\ln\Big(1+\sum_{i=1}^{m}x_i\Big).$$

Proof. Notice that for any $i \geq 1$:

$$\frac{x_i}{1+\sum_{j=1}^{i-1}x_j}=\int_{1+\sum_{j=1}^{i-1}x_j}^{1+\sum_{j=1}^{i}x_j}\frac{1}{1+\sum_{j=1}^{i-1}x_j}\,dx\geq\int_{1+\sum_{j=1}^{i-1}x_j}^{1+\sum_{j=1}^{i}x_j}\frac{1}{x}\,dx.$$

Hence:

$$\sum_{i=1}^{m}\frac{x_i}{1+\sum_{j=1}^{i-1}x_j}\geq\sum_{i=1}^{m}\int_{1+\sum_{j=1}^{i-1}x_j}^{1+\sum_{j=1}^{i}x_j}\frac{1}{x}\,dx=\int_{1}^{1+\sum_{j=1}^{m}x_j}\frac{1}{x}\,dx$$
$$=\ln\Big(1+\sum_{j=1}^{m}x_j\Big).$$ ◀

4.2 The "Carefully-Normalized" Read Potential Lemma

Our lower bound is based on the following key lemma, which intuitively provides a lower bound over the sum of the read potentials of registers written to in a solo execution e by processor p. Importantly, the read potentials are carefully normalized by, roughly, the probability that these registers are written to by other processors in some other executions.

▶ **Lemma 10.** *Let $e \in S(p)$ be a solo execution of processor p, and $\mathcal{TR}(e)$ be its trace. Then, $\sum_{r\in\mathcal{TR}(e)}\frac{\rho(r)}{1+\sum_{q\in P\setminus\{p\}}\sum_{e_q\in S(q)\cap\mathcal{B}(r)}\Pr[e_q]}\geq\ln n$.*

The rest of this sub-section will be dedicated to proving this lemma. Specifically, we prove the following two claims in the context of the theorem, i.e. for a fixed solo execution e of processor p. We expand $\mathcal{TR}(e)$ as $u_1^e, u_2^e, \ldots, u_{|\mathcal{TR}(e)|}^e$.

▷ Claim 11. $\sum_{i=1}^{|\mathcal{TR}(e)|}\frac{\rho(u_i^e)}{1+\sum_{j=1}^{i-1}\rho(u_j^e)}\geq\ln n$.

Proof. Applying Lemma 9 to positive real numbers $\rho(u_1^e), \rho(u_2^e), \ldots, \rho(u_{|\mathcal{TR}(e)|}^e)$, we get:

$$\sum_{i=1}^{|\mathcal{TR}(e)|} \frac{\rho(u_i^e)}{1 + \sum_{j=1}^{i-1} \rho(u_j^e)} \geq \ln\left(1 + \sum_{j=1}^{|\mathcal{TR}(e)|} \rho(u_j^e)\right).$$

By Lemma 7, $\sum_{j=1}^{|\mathcal{TR}(e)|} \rho(u_j^e) \geq n - 1$, completing the proof. ◁

Next, we prove the following extension:

▷ Claim 12. Under the above notation, we have

$$\sum_{j=1}^{i-1} \rho(u_j^e) \geq \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q) \cap \mathcal{B}(u_i^e)} \Pr[e_q].$$

Proof. Let us substitute the definition of ρ. We want to prove that:

$$\sum_{j=1}^{i-1} \sum_{q \in \mathcal{P}} \sum_{e_q \in S(q) \cap \mathcal{A}(u_j^e)} \Pr[e_q] \geq \sum_{q \in P \setminus \{p\}} \sum_{e_q \in S(q) \cap \mathcal{B}(u_i^e)} \Pr[e_q].$$

Both the left and right-hand sides of this expression contain the sum of probabilities of certain solo executions. On the right hand side, (the probability of) any execution e_q of processor $q \in P \setminus \{p\}$ can appear at most once. This is not necessarily true for the left hand side due to the outer summation. Therefore, we only need to show that for any e_q whose probability $\Pr[e_q]$ is included in the summation on the right hand side, $\Pr[e_q]$ is also included in the summation on the left hand side – in other words, there exists $1 \leq j \leq i - 1$, such that $\text{read}(u_j^e) \in e_q$.

We prove this fact by contradiction. Suppose processor $q \in P \setminus \{p\}$ has a solo execution $e_q \in S(q)$ such that register u_i^e is written to during e_q, but no register among $u_1^e, \ldots u_{i-1}^e$ are read (which are all registers written prior to u_i^e in p's solo execution e). Now consider a combined execution of p and q, which consists of p running as in e until it becomes poised to write register u_i^e – crucially, please note that so far p has actually executed solo. From this point, we consider processor q executing identically as it runs solo in execution e_q. This is possible because the only registers written to so far the system are $u_1^e, \ldots, u_{i-1}^e$, which q does not read in e_q. As a result, q will write to register u_i^e, after which we can immediately allow p to also write to u_i^e. This implies that two processors write to the same register during the same execution, and contradicting the SWMR property. ◁

Then, Lemma 10 follows by combining Claim 11, Claim 12 and the definition of trace.

4.3 Completing the Lower Bound Proof

We now finally proceed to proving the following theorem:

▶ **Theorem 13.** *Any randomized leader election protocol in asynchronous shared-memory with SWMR registers has* $\ln n$ *worst-case expected solo step complexity.*

Proof. We start by summing up inequalities given by Lemma 10 for all processors, and their solo executions:

$$
\begin{aligned}
n \ln n &\le \sum_{p\in\mathcal{P}} \sum_{e\in S(p)} \Pr[e] \left(\sum_{r\in\mathcal{TR}(e)} \frac{\rho(r)}{1+\sum_{q\in P\setminus\{p\}} \sum_{e_q\in S(q)\cap\mathcal{B}(r)} \Pr[e_q])} \right) \\
&\le \sum_{p\in\mathcal{P}} \sum_{e\in S(p)} \Pr[e] \left(\sum_{r\in\mathcal{TR}(e)} \frac{\rho(r)}{\sum_q \sum_{e_q\in S(q)\cap\mathcal{B}(r)} \Pr[e_q])} \right), \qquad (2)
\end{aligned}
$$

where in the last step we used that $\sum_{e_p\in S(p)\cap\mathcal{B}(r)} \Pr[e_p] \le \sum_{e_p\in S(p)} \Pr[e_p] = 1$. Hence, by using $\sum_q \sum_{e_q\in S(q)\cap\mathcal{B}(r)} \Pr[e_q] = \gamma(r)$ and Lemma 8 we get that

$$
n \ln n \le \sum_{p\in\mathcal{P}} \sum_{e\in S(p)} \Pr[e] \left(\sum_{r\in\mathcal{TR}(e)} \frac{\rho(r)}{\gamma(r)} \right) = \sum_{r\in\mathcal{R}} \rho(r).
$$

Note that $\rho(r)$ is the lower bound on the expected number of total reads from register r. Hence, since the expected number of total reads is at least $n \ln n$, there must exist a processor which performs at least $\ln n$ reads in expectation. ◀

4.4 Extension for Bounded Concurrent-Write Contention

We now extend our result to the case where the maximum number of processors which may be poised to write concurrently to a register, which we defined as the concurrent-write contention, is bounded. Specifically, suppose that, in any execution, at most $\kappa \ge 1$ different processors may be poised to write to the same register. We preserve the notation from the previous section. Upon close examination, notice that Lemma 7 and Lemma 8 still hold in this MWMR model, as well as Claim 11, since they do not employ the SWMR property. (By contrast, Claim 12 no longer holds for $\kappa > 1$.) We therefore continue to use only the above results.

We will prove a $\frac{\ln n}{\kappa^2}$ lower bound on the expected solo step complexity, under the above assumptions on κ. As before, let $\mathcal{TR}(e) = u_1^e, u_2^e, \ldots, u_{|\mathcal{TR}(e)|}^e$ be the trace of execution e.

▶ **Lemma 14.** *We have that* $n \ln n \le \sum_{r\in\mathcal{R}} \rho(r) \sum_{p\in\mathcal{P}} \sum_{e\in S(p)\cap\mathcal{B}(r)} \frac{Pr[e]}{1+\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)}$.

Proof. The proof is similar to the proof of Theorem 13, but using Claim 11 directly instead of Lemma 10. Specifically, we start by summing up the inequalities resulting from Claim 11 for all processors and solo executions:

$$
\begin{aligned}
n \ln n &\le \sum_{p\in\mathcal{P}} \sum_{e\in S(p)} \Pr[e] \left(\sum_{i=1}^{|\mathcal{TR}(e)|} \frac{\rho(u_i^e)}{1+\sum_{j=1}^{i-1} \rho(u_j^e)} \right) \\
&= \sum_{r\in\mathcal{R}} \rho(r) \sum_p \sum_{e\in S(p)\cap\mathcal{B}(r)} \frac{Pr[e]}{1+\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)}.
\end{aligned}
$$

The last equality follows by re-arranging terms to be grouped by register instead of by processor. Note that this is similar to the proof of Lemma 8. However, in this case the resulting equation cannot be simplified further, since, unlike $\gamma(u_i^e)$, the denominator term $1 + \sum_{j=1}^{i-1} \rho(u_j^e)$ also depends on the execution e. ◀

For any register r, we call the set of processors G *a poise set* for r if:

- G contains $g := |G|$ solo executions of different processors, i.e. $G = \{e_1, e_2, \ldots, e_g\}$, such that $r \in \mathcal{TR}(e_i)$, $e_i \in S(p_i)$ and $p_i \neq p_j$ for $i \neq j$.
- Let e_i' be the prefix of e_i up to and including p_i's first write step to the register r. There exists *a combined execution* e by processors $p_1, \ldots p_g$, such that at the end of e all processors p_i have written to r. Moreover, e is indistinguishable from e_i' to p_i (i.e. p_i takes steps as in e_i and does not read anything written by $p_{j \neq i}$ until it writes to r).

As g processors can be poised to write to r in the combined execution, no poise set can have size $> \kappa$.

▶ **Lemma 15.** *Let $E \subseteq \mathcal{B}(r)$ be a set of solo executions. Let k be the maximum size of a poise set for register r among executions in E. Then, there exists a subset of executions $H(E) \subseteq E$, such that:*

- $\sum_{e \in H(E)} \frac{Pr[e]}{1+\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} \leq k$;
- *Every poise set for register r among executions in $E \setminus H(E)$ has size at most $k-1$.*

Proof. Let $\beta = \min_{p_1,\ldots,p_k} \sum_{q \notin \{p_1,\ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e]$, i.e. the sum of probabilities of executions in E, excluding executions by k processors. We define the set H as follows:

$$H(E) = \Big\{ e \mid \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e) \geq \frac{\beta}{k} \Big\}.$$

So, an execution e is included in $H(E)$ if the sum of read potentials of registers written prior to r in e is lower bounded by a term that depends on k. Notice that for $k = 1$ this is analogous to the condition in Claim 12.
The parameter β satisfies the following useful property:

$$\begin{aligned} k + \beta &= k + \min_{p_1,\ldots,p_k} \sum_{q \notin \{p_1,\ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e] = \min_{p_1,\ldots,p_k} \Big(k + \sum_{q \notin \{p_1,\ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e]\Big) \\ &\geq \min_{p_1,\ldots,p_k} \Big(\sum_{q \in \{p_1,\ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e] + \sum_{q \notin \{p_1,\ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e]\Big) \qquad (3) \\ &= \min_{p_1,\ldots,p_k} \Big(\sum_{q} \sum_{e \in E \cap S(q)} Pr[e]\Big) = \sum_{q} \sum_{e \in E \cap S(q)} Pr[e] = \sum_{e \in E} Pr[e]. \end{aligned}$$

where in (3) we have used that, from the definition of S, $\sum_{e \in S(p_i)} \Pr[e] = 1$.
Using this property, we get:

$$\begin{aligned} \sum_{e \in H(E)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} &\leq \sum_{e \in H(E)} \frac{Pr[e]}{1 + \frac{\beta}{k}} = k \sum_{e \in h(E)} \frac{Pr[e]}{k+\beta} \\ &\leq k \frac{\sum_{e \in h(E)} Pr[e]}{\sum_{e \in E} Pr[e]} \leq k. \end{aligned}$$

This proves the first part of the lemma. We prove the second part of the lemma by contradiction. Suppose there is a poise set $G = \{e_1, e_2, \ldots e_k\}$ for register r among executions in $E \setminus H(E)$, where e_i is an execution of processor p_i.

Consider any execution $e' \in E \cap S(q)$ for $q \notin \{p_1, \ldots, p_k\}$. Execution e' must read one of the registers written during some time step before the point when r is written in e_i. Otherwise, e', and more precisely, the prefix of e' up to the write to r, can be added at the end of G's interleaved execution, implying that $G \cup \{e'\}$ would be a poise set of size $k+1$ among executions in E, which does not exist by definition. Hence:

$$\sum_{e \in G} \Big(\sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e) \Big) \geq \sum_{q \notin \{p_1, \ldots p_k\}} \sum_{e \in E \cap S(q)} Pr[e] \geq \beta.$$

Notice how this generalizes Claim 12: we can now apply the pigeonhole principle to $|G| = k$ terms on the left hand side. We get that for some i, $e_i \in H(E)$, giving the desired contradiction, specifically, that G consists of executions from $E \setminus H(E)$ only. This completes the proof of the Lemma. ◀

We are now ready to prove the main result of this section.

▶ **Theorem 16.** *Any randomized leader election protocol in asynchronous shared-memory has $\frac{\ln n}{\kappa^2}$ worst-case expected solo step complexity, when κ is the maximum number of processors which may be poised to write concurrently to the same register.*

Proof. Fix a register r. We start by applying Lemma 15 to the set $\mathcal{B}(r)$ and the maximum poise set size of κ. Let E_1 be the resulting subset of executions $H(\mathcal{B}(r))$, and $k_1 \leq \kappa$ be the maximum size of a poise set among executions in $\mathcal{B}(r) \setminus E_1$. Next, we apply Lemma 15 again to $\mathcal{B}(r) \setminus E_1$, and define $E_2 = H(\mathcal{B}(r) \setminus E_1)$, and $k_2 < k_1$ as the maximum size of a poise set among executions in $\mathcal{B}(r) \setminus (E_1 \cup E_2)$. The next application of Lemma 15 will be to $\mathcal{B}(r) \setminus (E_1 \cup E_2)$, defining $E_3 = H(\mathcal{B}(r) \setminus (E_1 \cup E_2))$ and $k_3 < k_2$. We repeat the process until some k_ℓ becomes 0, implying that the set of remaining executions $\mathcal{B}(r) \setminus (\cup_{t=1}^{\ell-1} E_t)$ is empty. Since $0 = k_\ell < k_{\ell-1} < \ldots \leq \kappa$, Lemma 15 will be applied at most κ times. Therefore we have obtained that:

$$\sum_p \sum_{e \in S(p) \cap \mathcal{B}(r)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} = \sum_{e \in \mathcal{B}(r)} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)}$$
$$= \sum_{t=1}^{\ell-1} \sum_{e \in E_t} \frac{Pr[e]}{1 + \sum_{j=1}^{\xi_r(e)-1} \rho(u_j^e)} \leq \kappa^2,$$

as there are at most κ terms, each of which is upper bounded by κ, by Lemma 15.

Combined with Lemma 14, this gives $\sum_r \rho(r) \geq \frac{n \ln n}{\kappa^2}$. By the pigeonhole principle, some processor must perform at least $\frac{\ln n}{\kappa^2}$ reads in expectation, over its solo executions. ◀

5 A Complementary Upper Bound for Weak Leader Election

It is interesting to consider whether the lower bound approach can be further improved to address the MWMR model under n-concurrent write contention. This is not the case for the specific definition of the weak leader election problem we consider (Definition 1), and to which the lower bound applies. To establish this, it suffices to notice that the classic *splitter* construction of Lamport [25] solves weak leader election for n processes, *in constant time*, by leveraging MWMR registers with maximal (concurrent) write contention n.

Please recall that this construction, restated for convenience in Figure 1, uses two MWMR registers. Given a splitter, we can simply map the *stop* output to *win*, and the *left* and *right* outputs to *lose*. In this case, it is immediate to show that the splitter ensures the following:

1. a processor will always return *win* in a solo execution, and
2. no two processes may return *win* in the same execution.

```
shared data:
atomic register race, big enough to hold an id, initially ⊥
atomic register door, big enough to hold a bit, initially open
procedure splitter(id)
    race ← id
    if door = closed then
        return right
    door ← closed
    if race = id then
        return stop
    else
        return down
```

Figure 1 The classic Lamport splitter [25], restated following [26, 6].

This matches the requirements of the *weak leader election* problem, but not of *test-and-set* objects generally, as this algorithm has contended executions in which all processors return *lose*, which is also impractical.

One may further generalize this approach by defining κ-splitter objects for $\kappa \geq 2$, each of which is restricted to κ participating processors (and thus also κ-concurrent write contention), and then arranging them in a complete κ-ary tree. We can then proceed similarly to tournament tree, to implement a weak leader election object. The resulting construction has $O(\log n / \log \kappa)$ step complexity in solo executions, suggesting that the dependency on κ provided by our argument can be further improved.

This observation suggests that the trade-off between step complexity and concurrent-write contention/worst-case stalls outlined by our lower bound may be the best one can prove for *weak leader election*, as this problem can be solved in constant time with MWMR registers, at the cost of linear worst-case stalls. At the same time, it shows that lower bound arguments wishing to approach the general version of the problem have to specifically leverage the fact that, even in contended executions, not all processors may return *lose*.

6 Conclusion

Overview. We gave the first tight logarithmic lower bounds on the solo step complexity of leader election in an asynchronous shared-memory model with single-writer multi-reader (SWMR) registers, for both deterministic and randomized algorithms. We then extended these results to registers with bounded concurrent-write contention $\kappa \geq 1$, showing a trade-off between the step solo complexity of algorithms, and their worst-case stall complexity. The approach admits additional extensions, and is tight in the SWMR case. The impossibility result is quite strong, in the sense that logarithmic time is required *over solo executions* of processors, and for a weak variant of leader election, which is not linearizable and allows processors to all return lose in in contended executions.

Future Work. The key question left open is whether sub-logarithmic upper bounds for strong leader election / test-and-set may exist, specifically by leveraging multi-writer registers, or whether the lower bounds can be further strengthened. Another interesting question is whether our approach can be extended to handle different cost metrics, such as remote memory references (RMRs).

References

1 Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 85–94, 1992.

2 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *International Symposium on Distributed Computing*, pages 97–109. Springer, 2011.

3 Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Rachid Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM (JACM)*, 61(3):1–51, 2014.

4 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th international conference on Distributed computing*, DISC'10, pages 94–108, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://portal.acm.org/citation.cfm?id=1888781.1888794.

5 Dan Alistarh, Rati Gelashvili, and Adrian Vladu. How to elect a leader faster than a tournament. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 365–374, 2015.

6 James Aspnes. Notes on theory of distributed systems. *arXiv preprint*, 2020. arXiv:2001.04235.

7 James Aspnes, Keren Censor-Hillel, Hagit Attiya, and Danny Hendler. Lower bounds for restricted-use objects. *SIAM Journal on Computing*, 45(3):734–762, 2016.

8 Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):1–26, 2008. doi:10.1145/1411509.1411510.

9 Hagit Attiya and Faith Ellen. Impossibility results for distributed computing. *Synthesis Lectures on Distributed Computing Theory*, 5(1):1–162, 2014.

10 James E Burns and Nancy A Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

11 Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

12 Aryaz Eghbali and Philipp Woelfel. An almost tight rmr lower bound for abortable test-and-set. *arXiv preprint*, 2018. arXiv:1805.04840.

13 Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

14 George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $O(\sqrt{n})$ space bound for obstruction-free leader election. In *International Symposium on Distributed Computing*, pages 46–60. Springer, 2013.

15 George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. Test-and-set in optimal space. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, pages 615–623, 2015.

16 George Giakkoupis and Philipp Woelfel. Efficient randomized test-and-set implementations. *Distributed Computing*, 32(6):565–586, 2019.

17 Wojciech Golab, Danny Hendler, and Philipp Woelfel. An o(1) rmrs leader election algorithm. *SIAM Journal on Computing*, 39(7):2726–2760, 2010.

18 Danny Hendler and Nir Shavit. Operation-valency and the cost of coordination. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 84–91, 2003.

19 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.

20 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

21 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

22 Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 201–210, 1998.

23 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

24 Yong-Jik Kim and James H Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

25 Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, 1987.

26 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25:1–39, 1995.

27 Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 91–97, New York, NY, USA, 1977. ACM. doi:10.1145/800105.803398.

28 Eugene Styer and Gary L Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 177–191, 1989.

29 John Tromp and Paul Vitányi. Randomized two-process wait-free test-and-set. *Distrib. Comput.*, 15(3):127–135, 2002. doi:10.1007/s004460200071.

30 Jae-Heon Yang and James H Anderson. Time bounds for mutual exclusion and related problems. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*, pages 224–233, 1994.

Deterministic Distributed Algorithms and Lower Bounds in the Hybrid Model

Ioannis Anagnostides ✉
Department of Computer Engineering, National Technical University of Athens, Greece

Themis Gouleakis ✉
Max Planck Institute for Informatics, Saarbrücken, Germany

Abstract

The HYBRID model was recently introduced by Augustine et al. [6] in order to characterize from an algorithmic standpoint the capabilities of networks which combine multiple communication modes. Concretely, it is assumed that the standard LOCAL model of distributed computing is enhanced with the feature of all-to-all communication, but with very limited bandwidth, captured by the node-capacitated clique (NCC). In this work we provide several new insights on the power of hybrid networks for fundamental problems in distributed algorithms.

First, we present a deterministic algorithm which solves any problem on a sparse n-node graph in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds of HYBRID, where the notation $\widetilde{\mathcal{O}}(\cdot)$ suppresses polylogarithmic factors of n. We combine this primitive with several sparsification techniques to obtain efficient distributed algorithms for general graphs. Most notably, for the all-pairs shortest paths problem we give deterministic $(1+\epsilon)$- and $\log n/\log\log n$-approximate algorithms for unweighted and weighted graphs respectively with round complexity $\widetilde{\mathcal{O}}(\sqrt{n})$ in HYBRID, closely matching the performance of the state of the art randomized algorithm of Kuhn and Schneider [33]. Moreover, we[1] make a connection with the Ghaffari-Haeupler framework of low-congestion shortcuts [21], leading – among others – to a $(1+\epsilon)$-approximate algorithm for Min-Cut after $\mathcal{O}(\text{polylog}(n))$ rounds, with high probability, even if we restrict local edges to transfer $\mathcal{O}(\log n)$ bits per round. Finally, we prove via a reduction from the set disjointness problem that $\widetilde{\Omega}(n^{1/3})$ rounds are required to determine the radius of an unweighted graph, as well as a $(3/2-\epsilon)$-approximation for weighted graphs. As a byproduct, we show an $\widetilde{\Omega}(n)$ round-complexity lower bound for computing a $(4/3-\epsilon)$-approximation of the radius in the broadcast variant of the congested clique, even for unweighted graphs.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Computing, Hybrid Model, Sparse Graphs, Deterministic Algorithms, All-Pairs Shortest Paths, Minimum Cut, Radius

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.5

Related Version *Full Version*: https://arxiv.org/abs/2108.01740

Acknowledgements We are indebted to Christoph Lenzen for carefully reviewing an earlier draft of our work, and proposing several improvements and interesting directions. Specifically, he suggested derandomizing Proposition 7 via the Garay-Kutten-Peleg algorithm, while he also pointed out the connection with low-congestion shortcuts, leading to the results of Section 4. We are also very grateful to the anonymous reviewers at DISC for carefully reviewing this paper, and for indicating many corrections and ways to improve the exposition. We are particularly thankful to a reviewer for providing very detailed arguments which strengthened our results in Section 3.4. All errors remain our own.

[1] See the acknowledgments.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 5; pp. 5:1–5:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Hybrid networks have found numerous applications in real-life computer systems. Indeed, leveraging different communication modes has substantially reduced the complexity and has improved the efficiency of the system, measured in terms of the energy consumption, the latency, the number of switching links, etc. For instance, hybrid architectures have been extensively employed in data centers, augmenting the traditional electrical switching architecture with optical switches in order to establish direct connections [11, 17, 51]. Another notable example is the 5G standard, which enhances the traditional cellular infrastructure with device-to-device (D2D) connections in order to guarantee very low latency among communication users (see [50, 29, 37, 41], and references therein).

Despite the central role of hybrid architectures in communication systems, a rigorous investigation of their potential has only recently began to formulate in the realm of distributed algorithms. In particular, Augustine et al. [6] proposed HYBRID, a model which combines the extensively-studied *local* (LOCAL) [38, 44] model with the recently introduced *node-capacitated clique* (NCC) [5]. The former model captures the *locality* of a given problem – nodes are able to exchange messages of arbitrary size but only with adjacent nodes, while the latter model – which enables all-to-all communication but with severe capacity restrictions for every node – addresses the issue of *congestion*; these constitute the main challenges in distributed computing. From a practical standpoint, the local network captures the capabilities of *physical* networks, wherein dedicated edges (e.g. cables or optical fibers) offer large bandwidth and high efficiency, but lack flexibility as they cannot be dynamically adapted by the nodes. In contrast, the global mode relates to *logical* networks, which are formed as an overlay over a shared physical network [18]; here the feature of all-to-all communication comes at the cost of providing very limited throughput.

In this work we follow the recent line of research [6, 33, 18, 9, 25, 10] which endeavors to explore from a theoretical standpoint the power of hybrid models in distributed computing; specifically, the main issue that arises is whether combining two different communication modes offers a substantial improvement over each mode separately. This question is answered in the affirmative for a series of fundamental problems in distributed algorithms, while we also provide some hardness results mainly based on well-established communication-complexity lower bounds.

1.1 Contributions & Techniques

Sparsification

First, we consider the design of HYBRID algorithms in *sparse* graphs – i.e. the average degree is $\operatorname{polylog} n$, where n represents the number of communication entities in the graph. We prove the following general result:

▶ **Theorem 1.** *Consider a graph $G = (V, E, w)$ with $|E| = \widetilde{\mathcal{O}}(n)$. There exists a deterministic distributed algorithm such that every node learns the entire topology of the graph in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds of* HYBRID.

As a warm-up, we first provide a *randomized* HYBRID algorithm so that every node learns the topology in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds. More importantly, we also derandomize this communication pattern (Theorem 1), leading along the way to a derandomization of the token dissemination protocol of Augustine et al. [6], which is one of their main communication primitives. Specifically, we first employ the Garay-Kutten-Peleg algorithm [20] in order to construct a "balanced" partition of the nodes, so that every cluster has "small" weak diameter. Then, we present several deterministic subroutines which allow to disseminate the composition of the clusters, perform load balancing, and finally broadcast the topology to the entire network within the desired round complexity.

Naturally, our guarantee for sparse graphs has an independent interest given that most communication networks of practical interest are very sparse [36]; the canonical example typically cited is the Internet [40]. Nonetheless, we leverage several *sparsification* techniques in order to design distributed algorithms for general graphs. In particular, we first employ a deterministic *multiplicative spanner* algorithm [23, 49] to obtain a $\log n / \log \log n$-approximation for the weighted *all-pairs shortest paths* (APSP) problem in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds. For unweighted graphs we leverage the recent deterministic *near-additive spanner* due to Elkin and Mater [16], leading to a $(1+\epsilon)$-approximate algorithm for APSP, for any constant $\epsilon > 0$. Although this does not quite reach the performance of the state of the art algorithm of Kuhn and Schneider [6], which yields an exact solution for weighted graphs with asymptotically the same round-complexity (modulo polylogarithmic factors), we stress that our algorithms are *deterministic*.

Moreover, we use *cut sparsifiers* in order to provide near-optimal algorithms for any cut-related problem in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds. Here it is important to point out that our algorithmic scheme "Sparsify & Conquer" is primarily meaningful when the output requirement is *global*. For example, for the Min-Cut problem, if we require that every node knows a cut at the end of the distributed algorithm, we show an $\widetilde{\Omega}(\sqrt{n})$ round-complexity lower bound for any non-trivial approximation based on a technical lemma in [6]. However, in many settings this approach may disseminate an overly amount of information. Indeed, under the usual requirement that each node has to know its "side" on the cut, we establish exponentially faster algorithms.

Simulating CONGEST-based Algorithms

This accelerated algorithm for Min-Cut is obtained through a connection with the concept of *low-congestion shortcuts*, due to Ghaffari and Haeupler [21]. Specifically, in this framework the performance-guarantee for a problem is parameterized in terms of the number of rounds required to solve the standard part-wise aggregation problem. A fascinating insight of Ghaffari and Haeupler [21] is that more "structured" topologies (e.g. planar graphs) enable faster algorithms for solving such problems, bypassing some notorious lower bounds under general graphs. Our observation is that a limited amount of global power, in the form of NCC, interacts particularly well with this line of work since NCC offers very fast primitives for the part-wise aggregation problem. As a result, this connection leads to the following result:

▶ **Theorem 2.** *There exists an $\mathcal{O}(\text{polylog}(n))$-round algorithm for $(1+\epsilon)$-approximate Min-Cut in* $\mathsf{CONGEST} + \mathsf{NCC}$.

Note that this guarantees applies even if local edges are restricted to transfer only $\mathcal{O}(\log n)$ bits per round, i.e. the local network is modeled with CONGEST instead of LOCAL. Another notable corollary of this connection is an approximate single-source shortest paths algorithm (Corollary 21) based on a result by Haeupler and Li [26], coming close to the algorithm of Augustine et al. [6] under the substantially more powerful HYBRID model. We also present another simulation argument, which in a sense eliminates the dependence of the performance of a CONGEST algorithm on the hop-diameter through an appropriate augmentation of the graph with global edges (see Proposition 22).

Distance Computation Tasks

Finally, we focus on distance computation tasks, and in particular, the complexity of determining the *radius* and the *diameter* of the underlying graph – the smallest and the largest of the eccentricities respectively. For the former, we show the following result:

▶ **Theorem 3.** *For any $\epsilon \in (0, 1/2]$, determining a $(3/2 - \epsilon)$-approximation for the radius of a weighted graph with probability $2/3$ requires $\widetilde{\Omega}(n^{1/3})$ rounds of* HYBRID*. For unweighted graphs, determining the radius requires $\widetilde{\Omega}(n^{1/3})$ rounds of* HYBRID*.*

This limitation applies for any randomized distributed algorithm even if we allow a substantial probability of failure (i.e. Monte Carlo algorithms), and/or public (common) randomness. We should point out that our lower bound for unweighted graphs matches the known upper bound for *approximate* radius, as the authors in [9] provide a $(1 + \epsilon)$-approximation for all the unweighted eccentricities in $\widetilde{\mathcal{O}}(n^{1/3})$ rounds of HYBRID, for any constant $\epsilon > 0$. Our theorem also supplements the hardness result of Kuhn and Schneider [33] who established analogous lower bounds for the diameter.

More precisely, we give a suitable *dense* gadget graph whose edges correspond to the input-strings of two players endeavoring to solve the set disjointness problem. Then, we show that there is a gap in the value of the radius depending on whether the input of the two players is *disjoint*. Our construction uses a *bit-gadget*, a component introduced in [1] (see also [2]) in order to show a linear lower bound for determining the radius in CONGEST, even for sparse graphs. Nonetheless, our reduction has several differences given that the source of the communication bottleneck is quite different in CONGEST (where it suffices to induce a bottleneck in the *communication cut* between the two players) compared to a model with all-to-all communication. As a result, we first prove an $\widetilde{\Omega}(n)$ round-complexity lower bound for determining a $(4/3 - \epsilon)$-approximation of the radius in the *broadcast* variant of the *congested clique* (BCC), for any $\epsilon \in (0, 1/3]$, even for unweighted graphs; we consider this result to be of independent interest. Next, with minor modifications in the construction we show Theorem 3. These results require simulation arguments, establishing that Alice and Bob can indeed employ (or simulate) the communication pattern of the distributed algorithm in order to solve the set disjointness problem. In this context, for the HYBRID model we make use of the simulation argument of Kuhn and Schneider [33].

Finally, for the weighted diameter the state of the art algorithm in HYBRID simply performs a Dijkstra search from an arbitrary source node and returns as the estimation the eccentricity (i.e. the largest distance) of the source node [9]; an application of the triangle inequality implies that this algorithm yields a 2-approximation of the actual diameter. We make a step towards improving this approximation ratio. Specifically, we show that for graphs with small degrees ($\Delta = \mathcal{O}(\operatorname{polylog} n)$) we can obtain a 3/2-approximation of the diameter with asymptotically the same round-complexity, namely $\widetilde{\mathcal{O}}(n^{1/3})$ rounds. This result is based on the sequential algorithm of Roditty and Vassilevska W. [48]. Our contribution is to establish that their algorithm can be substantially parallelized in HYBRID; this is shown by employing some machinery developed in [9] for solving in parallel multiple single-source shortest paths problems.

Due to space constraints, most of the proofs, as well as additional results are presented in the full version of this paper [3].

1.2 Related Work

As we explained in our introduction, the HYBRID model was only recently introduced by Augustine, Hinnenthal, Kuhn, Scheideler, and Schneider [6]. Specifically, they developed several useful communication primitives in order to tackle distance computation tasks;

Table 1 An overview of our main results; it is assumed that $\epsilon > 0$ is an arbitrarily small constant.

Problem	Variant	Approximation	Model	Complexity	Technique
Deterministic APSP	Unweighted	$1+\epsilon$	HYBRID	$\widetilde{\mathcal{O}}(\sqrt{n})$	Sparsification: [49, 16]
	Weighted	$\log n / \log \log n$			
MST	Weighted	Exact	CONGEST + NCC	$\mathcal{O}(\log^2 n)$	Shortcuts: [21, 26]
Min-Cut	Weighted	$1+\epsilon$		$\mathcal{O}(\text{polylog}(n))$	
SSSP	Weighted	$\text{polylog}(n)$		$\widetilde{\mathcal{O}}(n^{\epsilon})$	
Radius	Unweighted	$4/3-\epsilon$	BCC	$\widetilde{\Omega}(n)$	Set Disjointness: [28, 1, 33]
	Unweighted	Exact	HYBRID	$\widetilde{\Omega}(n^{1/3})$	
	Weighted	$3/2-\epsilon$			

most notably, for the SSSP problem they established a $(1+o(1))$-approximate solution in $\widetilde{\mathcal{O}}(n^{1/3})$ rounds, while they also presented an algorithm with round complexity $\widetilde{\mathcal{O}}(\sqrt{n})$ for approximately solving the weighted APSP problem with high probability.[2] Their lower bound for the APSP problem was matched in a subsequent work by Kuhn and Schneider [33], showing that $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds suffice in order to *exactly* solve APSP. They also presented an $\widetilde{\Omega}(n^{1/3})$ lower bound for determining the diameter based on a reduction from the two-party set disjointness problem.

Moreover, Censor-Hillel et al. [9] improved several aspects of the approach in [6], showing how to exactly solve multiple SSSP problems in $\widetilde{\mathcal{O}}(n^{1/3})$ rounds; they also presented near-optimal algorithms for approximating all the eccentricities in the graph. For the approximate SSSP problem an improvement over the result in [6] was recently achieved by Censor-Hillel et al. [10], obtaining a $(1+\epsilon)$-approximate algorithm in $\widetilde{\mathcal{O}}(n^{5/17})$ rounds of HYBRID, for a sufficiently small constant $\epsilon > 0$. More restricted families of graphs (e.g. very sparse graphs or *cactus* graphs) were considered by Feldmann et al. [18], establishing an exponential speedup over some of the previous results even though they modeled the local network via CONGEST, which is of course substantially weaker than LOCAL. Finally, Götte et al. [25] provided several fast hybrid algorithms for problems such as connected components, spanning tree, and the maximal independent set.

The *node-capacitated clique* model (NCC) was recently introduced in [5]; it constitutes a much weaker – and subsequently much more realistic – model than the *congested clique* (CLIQUE) of Lotker et al. [39] in which every node can communicate with *any* other node (instead of only $\mathcal{O}(\log n)$ other nodes in NCC) with $\mathcal{O}(\log n)$-bit messages. Indeed, in CLIQUE a total of $\widetilde{\Theta}(n^2)$ bits can be transmitted in each round, whereas in NCC the cumulative broadcasting capacity is only $\widetilde{\Theta}(n)$ bits; as evidence for the power of CLIQUE we note that even slightly super-constant lower bounds would give new lower bounds in circuit complexity, as implied by a simulation argument in [15].

Reductions from communication complexity to distributed computing are by now fairly standard in the literature; see [45, 13, 19] and references therein. We also refer to [45, 13, 19] for reductions in the *broadcast* variant of CLIQUE where in each round every node can send the *same* $\mathcal{O}(\log n)$-bit message to all the nodes. Our construction for the radius is inspired by the gadget in [1], wherein the authors showed near-linear lower bounds for determining the radius in CONGEST, even for sparse networks. Finally, we refer to [22, 42, 24] for some of the state of the art technology for the Min-Cut problem.

[2] We will say that an event holds *with high probability* if it occurs with probability at least $1 - 1/n^c$ for some constant $c > 0$.

2 Preliminaries

We assume that the network consists of a set of n communication entities (e.g. processors) with $[n] \overset{\text{def}}{=} \{1, 2, \ldots, n\}$ the set of IDs, and a local communication *topology* given by a graph $G = (V, E, w)$. We will tacitly posit that G is *undirected*, unless explicitly stated otherwise; we also assume that for all $e \in E, w(e) \in \{1, 2, \ldots, W\}$, for some $W = \text{poly}(n)$. At the beginning each node knows the identifiers of each node in its neighborhood, but has no further knowledge about the topology of the graph. Communication occurs in *synchronous rounds*; in every round nodes have unlimited computational power[3] to process the information they posses. The *local* communication mode will be modeled with LOCAL, for which in each round every node can exchange a message of *arbitrary* size with its neighbors in G via the *local* edges. The *global* communication mode uses NCC for which in each round every node can exchange $\mathcal{O}(\log n)$-bit messages with up to $\mathcal{O}(\log n)$ arbitrary nodes via *global* edges. More broadly, one can parameterize hybrid networks by the number of bits λ that can be exchanged via local edges, and the number of bits γ that can be exchanged via the global mode. Interestingly, all standard models can be seen as instances of this general parameterization; namely, LOCAL : $\lambda = \infty, \gamma = 0$, CONGEST : $\lambda = \mathcal{O}(\log n), \gamma = 0$, CLIQUE : $\lambda = 0, \gamma = \mathcal{O}(n \log n)$,[4] NCC : $\lambda = 0, \gamma = \mathcal{O}(\log^2 n)$.

If the capacity of some channel is exceeded the corresponding nodes will only receive an *arbitrary* (potentially adversarially selected) subset of the information according to the capacity of the network, while the rest of the messages are dropped. The performance of a distributed algorithm is measured in terms of its *round-complexity* – the number of rounds required so that every node knows its part of the output; for randomized protocols it will suffice to reach the desired state with high probability. Finally, all of the derived round-complexity upper bounds in HYBRID should be thought of as having a minimum with the (hop) diameter of the network.

2.1 Useful Communication Primitives

A *distributive aggregate function* f maps a multiset $S = \{x_1, \ldots, x_N\}$ of input values to some value $f(S)$, such that there exists an aggregate function g so that for any multiset S and any partition $S_1, \ldots S_\ell, f(S) = g(f(S_1), \ldots, f(S_\ell))$; typical examples that we will use include Max, Min, and Sum. Now consider that we are given a distributive aggregate function f and a set $A \subseteq V$, so that every member of A stores *exactly* one input value. The *aggregate-and-broadcast* problem consists of letting every node in the graph learn the value of f evaluated at the corresponding input.

▶ **Lemma 4** ([5], Theorem 2.2). *There exists an algorithm in* NCC *which solves the aggregate-and-broadcast problem in* $\mathcal{O}(\log n)$ *rounds.*

In the (k, ℓ)-token dissemination problem (henceforth abbreviated as (k, ℓ)-TD) there are k (distinct) tokens (or messages), each of size $\mathcal{O}(\log n)$ bits, with every node initially having at most ℓ tokens. The goal is to guarantee that every node in the graph has collected all of the tokens.

▶ **Lemma 5** ([6], Theorem 2.1). *There exists a randomized algorithm in* HYBRID *which solves the* (k, ℓ)*-TD problem on connected graphs in* $\widetilde{\mathcal{O}}(\sqrt{k} + \ell)$ *rounds with high probability.*

[3] Nonetheless, we remark that most of our algorithms use a reasonable amount of computation.

[4] This follows from Lenzen's routing [34].

2.2 Communication Complexity

Most of our lower bounds are established based on the communication complexity of *set disjointness*, arguably the most well-studied problem in communication complexity (e.g., see [27, 43, 47]). More precisely, consider two communication parties – namely Alice and Bob – with infinite computational power. Every player is given a binary string of k-bits, represented with $x, y \in \{0,1\}^k$ respectively, and their goal is to determine the value of a function $f(x, y)$ by interchanging messages between each other. The players are allowed to use randomization, and the complexity is measured by the expected number of communication in the worst case [52]. For probabilistic protocols the players are required to give the right answer with some probability bounded away from $1/2$, i.e. to outperform random guessing; for concreteness, we assume that the probability of being correct should be $2/3$. It is also interesting to point out that common (public) randomness is allowed, with Alice and Bob sharing an infinite string of independent coin tosses.

In the set disjointness problem (DISJ_k) the two parties have to determine whether there exists $i \in [k]$ such that $x_i = y_i = 1$; in other words, if the inputs x and y correspond to subsets of a universe Ω, the problem asks whether the two subsets are disjoint – with a slight abuse of notation this will be represented with $x \cap y = \emptyset$. We will use the following celebrated result due to Kalyanasundaram and Schnitger [28].

▶ **Theorem 6** ([28]). *The randomized communication complexity of* DISJ_k *is* $\Omega(k)$.

3 Sparsification in Hybrid Networks

As a warm-up, we commence this section by presenting an $\widetilde{\mathcal{O}}(\sqrt{n})$ *randomized* protocol for solving *any* problem on *sparse* graphs in the HYBRID model. More importantly, we also present a *deterministic* algorithm with asymptotically the same round complexity, up to polylogarithmic factors. Next, we present several applications of this result in general graphs via distributed *sparsification* techniques.

3.1 Randomized Protocol

▶ **Proposition 7** (Randomized Hybrid Algorithm for Sparse Networks). *Consider an n-node (connected) graph $G = (V, E, w)$. There exists a randomized algorithm so that every node in V can learn the entire topology of the graph in $\widetilde{\mathcal{O}}(\sqrt{|E|})$ rounds of* HYBRID *with high probability.*

Sketch of Proof. Suppose that the lowest degree node incident to each edge $e \in E$ is responsible for disseminating the information about that edge. Then, each node will be responsible for at most $\sqrt{2|E|}$ edges since otherwise it would have at least $\sqrt{2|E|}$ neighbors with degree at least $\sqrt{2|E|}$, which is a contradiction. Thus, the result follows from Lemma 5. ◀

3.2 Deterministic Protocol

Before we proceed with our deterministic algorithm let us first recall that the *strong diameter* of a subset $C \subset V$ is the diameter of the subgraph induced by C; in contrast, the *weak diameter* of C is measured in the original graph. We will analyze and explain every step of the algorithm separately. We stress that the round-complexity in some steps has not been optimized since it would not alter the asymptotic running time of the protocol. Also note that in the sequel we use the words component and cluster interchangeably.

Algorithm 1 Deterministic HYBRID Algorithm for Sparse Networks.

```
Input: An $n$-node graph $G = (V, E)$ such that $|E| = \widetilde{\mathcal{O}}(n)$.
Output Requirement: Every node knows the entire topology of $G$.
1. Determine a partition of the nodes $V$ into $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_k$ via the
   Garay-Kutten-Peleg algorithm such that for all $i$,
     (i) the strong diameter of $\mathcal{C}_i$ is $\mathcal{O}(\sqrt{n})$;
     (ii) $|\mathcal{C}_i| \geq \sqrt{n}$.
2. Let $C_1, C_2, \dots, C_N = \textsc{Fragment}(\mathcal{C}_1, \dots, \mathcal{C}_k)$ such that for all $i$,
     (i) the weak diameter of $C_i$ is $\mathcal{O}(\sqrt{n})$;
     (ii) $\sqrt{n} \leq |C_i| < 2\sqrt{n}$.
3. Broadcast the IDs of the components' leaders.
4. Distribute all the $C_i$'s via the MatchingComponents subroutine.
5. Assign every edge on a component, and perform LoadBalancing.
6. Disseminate all the information.
```

Step 1. The first step of the algorithm partitions the set of nodes into a collection of connected components $\mathcal{C}_1, \dots, \mathcal{C}_k$, so that the minimum size is at least $\sqrt{n}$ and the strong diameter in every component is $\mathcal{O}(\sqrt{n})$; for simplicity we will assume that $\sqrt{n}$ is an integer. This step will be implemented with the standard Garay-Kutten-Peleg (GKP) algorithm [20, 35]. Specifically, GKP is an MST algorithm which operates in two phases; we will only need the first phase. The main idea is to gradually perform merges but in a "balanced" manner. More precisely, GKP maintains a set of components. In each iteration i every component with diameter at most 2^i determines the minimum-weight outgoing edge, which is subsequently added to a set of "candidates" edges. Then, the algorithm determines a maximal matching on this set, updating the components accordingly. If a component with diameter smaller than 2^i did not participate in the maximal matching, the algorithm automatically incorporates the edge that was selected by it. This process is repeated for $i = 0, 1, \dots, \lceil \log \sqrt{n} \rceil$, leading to a partition of V into $\mathcal{C}_1, \dots, \mathcal{C}_k$.

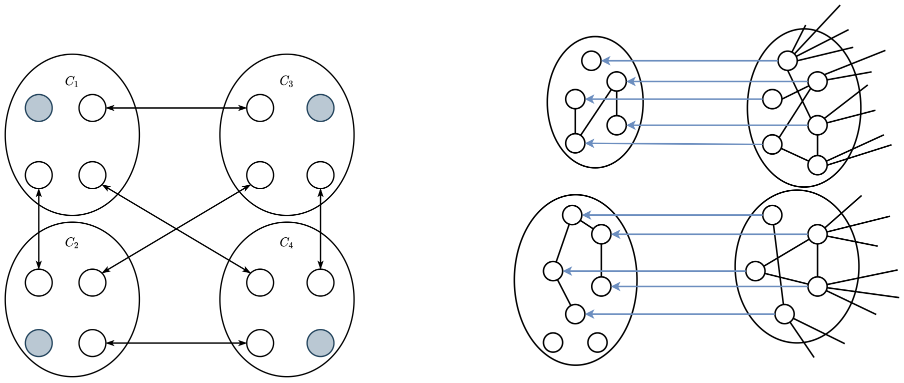
▶ **Lemma 8** ([35]). *At the end of the first phase of the* GKP *algorithm every component has strong diameter $\mathcal{O}(\sqrt{n})$, while every component has at least $\sqrt{n}$ nodes.*

This lemma verifies our initial claim for step 1. Moreover, note that the first phase of GKP can be implemented in $\mathcal{O}(\sqrt{n} \log^* n)$ rounds in CONGEST; naturally, in HYBRID we can substantially reduce the number of rounds, but this would not affect the overall asymptotic complexity as there is an inherent bottleneck in subsequent steps of the algorithm.

Step 2. The Fragment subroutine of the second step is particularly simple. If a component $\mathcal{C}_i$ is such that $|\mathcal{C}_i| < 2\sqrt{n}$ it remains intact. Otherwise, the component $\mathcal{C}_i$ is decomposed arbitrarily into disjoint fragments each of size between $\sqrt{n}$ and $2\sqrt{n}$. Let $C_1, \dots, C_N$ be the induced partition of V. By virtue of Lemma 8 we know that the *weak* diameter of every C_i is $\mathcal{O}(\sqrt{n})$, although note that the induced graph on C_i is potentially disconnected. This step is made to ensure that the components have roughly the same size, while it can be trivially implemented in $\mathcal{O}(\sqrt{n})$ rounds of LOCAL.

Step 3. We assume that every component has elected a leader, e.g. the node with the smallest ID. There are overall $N \leq \sqrt{n}$ IDs to be broadcast to the entire graph. This can be implemented with N (deterministic) broadcasts in NCC, which requires $\mathcal{O}(\sqrt{n} \log n)$ rounds.

Step 4. The purpose of this step is to ensure that every node knows the composition – i.e. the set of IDs – of every other cluster. To this end, the leader of every component C_i selects arbitrarily $N - 1 \leq |C_i| - 1$ *representative* nodes from C_i, and devises a (bijective) mapping from these nodes to all the other components; the leader also informs via the local network the corresponding nodes. Then, the protocol proceeds in rounds: In every iteration a single component interacts with all the others, and specifically, every representative node sends its ID to the leader of its assigned component. This is repeated for all the components, and after $\mathcal{O}(\sqrt{n})$ rounds every representative node will be matched with some node on its corresponding component; see Figure 1a. Having established this matching every node can disseminate through the global network the IDs of all the nodes in its own component to its assigned node. This process requires at most $2\sqrt{n}$ rounds since every component has size less than $2\sqrt{n}$, and every node participates in at most one matching. Finally, the composition (the set of IDs) of every component is revealed to each node after $\mathcal{O}(\sqrt{n})$ additional rounds of the local network.



(a) An example of MatchingComponents.

(b) LoadBalancing: Transferring load from overloaded to underloaded components. We have highlighted with blue "global" edges.

Step 5. First of all, every edge with incident nodes residing on the same component is assigned to the component of its endpoints. Otherwise, the edge is assigned to one of the components according to some deterministic rule; e.g. the component with the smaller ID. In this context, the *load* of every component is the number of edges it has to disseminate. Notice that the nodes of each component can learn every component's load in $\mathcal{O}(\sqrt{n})$ rounds. Initially, the load of each component is distributed uniformly within the nodes of the component, which requires $\mathcal{O}(\sqrt{n})$ rounds. The LoadBalancing mechanism works as follows: It splits the components into a set of "overloaded" components with load more than $2|E|/N$, and a set of "underloaded" components with load less than $|E|/N$; every other component does not need any further processing. In every iteration we map arbitrarily (e.g. the overloaded component with the smallest ID is mapped to the underloaded component with the smallest ID, and so on) overloaded components to underloaded ones, so that the mapping is one-to-one and maximal. Then, every assigned overloaded component transmits as much load as is required to its corresponding component via the global network (see Figure 1b) until one of the two becomes balanced – according to the previous notion. This can be performed in $\widetilde{\mathcal{O}}(1)$ by virtue of Step 4 (recall that $|E| = \widetilde{O}(n)$). Then, we remove any components that have been balanced and we proceed recursively for the remaining ones.

It is easy to see that this process requires at most N iterations since in every iteration we eliminate at least one component from requiring further balancing, while at the end of this step every component will have $\widetilde{\mathcal{O}}(\sqrt{n})$ load.

Step 6. The final step is fairly straightforward. First, observe that a single component can transfer its entire load to another component in $\widetilde{\mathcal{O}}(1)$ rounds via the global network; this follows because (i) every component has $\widetilde{\mathcal{O}}(\sqrt{n})$ load due to the load balancing step, and (ii) every component has by construction roughly $\sqrt{n}$ nodes. Assume that the components $C_1, \dots, C_N$ are sorted in ascending order with respect to their IDs. Then, at iteration i component C_j transfers its load to C_{r+1}, where $r = i + j \mod N$. This is repeated for $i = 0, 1, \dots, N-2$. It is easy to see that this deterministic protocol guarantees that (i) no *collisions* occur, and (ii) every component eventually receives the load from all other components. As we previously argued every such iteration requires $\widetilde{\mathcal{O}}(1)$ rounds in HYBRID. Thus, overall this step requires $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds, leading to the following conclusion:

▶ **Theorem 1.** *Consider a graph $G = (V, E, w)$ with $|E| = \widetilde{\mathcal{O}}(n)$. There exists a deterministic distributed algorithm such that every node learns the entire topology of the graph in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds of* HYBRID.

More broadly, our deterministic protocol can be used for any m-edge (connected) graph by forming clusters of size $\Theta(\sqrt{m})$ nodes, so that every node learns the topology after $\widetilde{\mathcal{O}}(\sqrt{m})$ rounds. Consequently, this leads to a derandomization of the token dissemination algorithm of Augustine et al. [6] in the regime $k \geq n$:

▶ **Proposition 9.** *There exists a deterministic algorithm which solves the (k, ℓ)-TD problem in $\widetilde{\mathcal{O}}(\sqrt{k})$ rounds of* HYBRID*, assuming that $k \geq n$.*

3.3 Distributed Sparsification Techniques

Before we present applications of our protocol for general graphs, we first review some basic sparsification techniques. The goal is to efficiently sparsify the graph in a distributed fashion, while approximately preserving some *structure* in the graph. We will use two fundamental notions of sparsifiers.

Spanners

The first structure one may wish to retain with sparsification is every pairwise distance in the graph. To this end, we will employ the notion of a graph *spanner*, a fundamental object in graph theory with numerous applications in distributed computing [46]. To be more precise, for a graph $G = (V, E)$, a subgraph H is an *α-stretch spanner* if every pairwise distance in H is at most an $\alpha \geq 1$ factor larger than the distance in G, without ever underestimating; i.e., for all $u, v \in G, d_G(u, v) \leq d_H(u, v) \leq \alpha \cdot d_G(u, v)$. Naturally, we desire spanners with small stretch and a limited number of edges. It is well-known that any n-node graph admits a $(2k-1)$-stretch spanner with $\mathcal{O}(n^{1+1/k})$ number of edges, while this trade-off is optimal conditioned on Erdős girth conjecture. In the distributed context, we will use the following result of Rozhon and Ghaffari:

▶ **Theorem 10** ([49]). *Consider an n-node weighted graph $G = (V, E)$. There exists a deterministic distributed algorithm in* CONGEST *which computes a $(2k-1)$-stretch spanner of size $\widetilde{\mathcal{O}}(kn^{1+1/k})$ in $\operatorname{polylog}(n)$ rounds.*

Besides *multiplicative* spanners, we will also use *near-additive* spanners. More precisely, a subgraph H of G is an (α, β)-stretch spanner if for all $u, v \in G, d_G(u,v) \leq d_H(u,v) \leq \alpha \cdot d_G(u,v) + \beta$; for $\beta = 0$ this recovers the previous notion of a multiplicative spanner. Moreover, for $\alpha = 1 + \epsilon$, for an arbitrarily small $\epsilon > 0$, the spanner is called near-additive. In this context, we will leverage the following recent result due to Elkin and Matar [16]:

▶ **Theorem 11** ([16]). *Consider an n-node unweighted graph $G = (V, E)$. For any constants $\epsilon \in (0, 1)$ and $\rho \in (0, 1/2)$, there is an algorithm in* CONGEST *which computes a $(1 + \epsilon, \beta)$-stretch spanner of G with $\widetilde{\mathcal{O}}(n)$ number of edges after $\mathcal{O}(\beta n^\rho)$ rounds, where $\beta = \mathcal{O}((\log \log n/\rho + 1/\rho^2)^{\log \log n + 1/\rho})$.*

Cut Sparsifiers

Another fundamental class of sparsifiers endeavors to approximately preserve the weight of every *cut* in the graph. Recall that for a subset of vertices $S \subset V$ we define

$$\operatorname{cut}_G(S) = \sum_{u \in S, v \in V \setminus S} w(u,v). \tag{1}$$

To this end, we will employ the sparsification algorithm developed by Koutis [32]. We should remark that the algorithm of Koutis actually returns a *spectral* sparsifier, which is a strictly stronger notion than a cut sparsifier [7], but we will not use this property here.

▶ **Theorem 12** ([32], Theorem 5). *Consider a graph $G = (V, E, w)$. There exists a distributed algorithm in* CONGEST *such that for any $\epsilon > 0$ outputs a graph $H = (V, \widehat{E}, \widehat{w})$ after $\widetilde{\mathcal{O}}(1/\epsilon^2)$ rounds such that (i) $(1 - \epsilon) \operatorname{cut}_H(S) \leq \operatorname{cut}_G(S) \leq (1 + \epsilon) \operatorname{cut}_H(S)$ for any $S \subset V$, and (ii) the expected number of edges in H is $\widetilde{\mathcal{O}}(n/\epsilon^2)$.*

3.4 Applications

Deterministic APSP

In the α-approximate *all-pairs shortest paths* problem every node $u \in V$ has to learn a value $d'(u, v)$ such that $d(u, v) \leq d'(u, v) \leq \alpha \cdot d(u, v)$, for all $v \in V$. In this context, we establish the following result:

▶ **Proposition 13** (Weighted APSP). *Consider an n-node weighted graph $G = (V, E)$. There exists a deterministic $\log n / \log \log n$-approximation algorithm for the APSP problem which runs in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds of* HYBRID.

For unweighted graphs we use near-additive spanners (Theorem 11) to improve upon the approximation ratio established for weighted graphs.

▶ **Proposition 14** (Unweighted APSP). *Consider an n-node unweighted graph $G = (V, E)$. For any constant $\epsilon \in (0, 1)$, there exists a deterministic $(1 + \epsilon)$-approximation algorithm for the APSP problem which runs in $\widetilde{\mathcal{O}}(\sqrt{n})$ rounds of* HYBRID.

Cut Problems

Moreover, we will leverage the distributed algorithm of Koutis in order to obtain efficient algorithms for cut-related problems in the HYBRID model. We wish to convey the robustness of our approach by presenting a guarantee for a series of cut problems. First, we recall the following: The *minimum cut* problem consists of identifying a partition of the vertices V into

S and $V \setminus S$ in order to minimize the weight of $\text{cut}_G(S)$; it admits an efficient centralized solution, for example, via Karger's celebrated algorithm [30]. Note that for an unweighted graph the minimum cut coincides with the *edge connectivity*. The *$s-t$ minimum cut* problem is similar to the minimum cut problem, but the nodes s and t are restricted to reside on different sets of the partition; see [8, 12]. Finally, in the *sparsest cut* problem we are searching for a partition $S, V \setminus S$ that minimizes the quantity $\text{cut}_G(S)/(|S| \cdot |V \setminus S|)$; it is known that the sparsest cut problem is $\mathcal{NP}$-hard [4, 31].

▶ **Proposition 15** (HYBRID Algorithms for Cut Problems). *For any n-node graph $G = (V, E, w)$ and for any $\epsilon \in (0, 2)$, we can compute with high probability a $(1+\epsilon)$-approximation in expected $\widetilde{\mathcal{O}}(\sqrt{n}/\epsilon + 1/\epsilon^2)$ rounds of* HYBRID *for the following problems: (i) the minimum $s-t$ cut, (ii) the minimum cut, and (iii) the sparsest cut.*

This approach is meaningful for cut-related problems once we impose a stronger output requirement. Namely, we guarantee that every node will know at the end of the distributed algorithm the entire composition of an approximate cut. In fact, for such an output requirement we can establish an almost-matching lower bound based on a technique developed in [6]:

▶ **Proposition 16.** *Determining a W/n-approximation for the minimum cut problem requires $\widetilde{\Omega}(\sqrt{n})$ rounds of* HYBRID*, where $W \geq n$ is the maximum edge-weight, assuming that every node has to know a cut at the end of the distributed algorithm.*

4 Simulating CONGEST Algorithms

The approach developed in the previous section is primarily meaningful for problems with a very demanding output requirement; for example, APSP, or cut-related problems for which nodes have to learn the exact composition of the cut. In contrast, in this section we will show that, for the minimum cut problem, we can obtain substantially faster algorithms when the nodes have to simply learn their "side" on the cut, which constitutes the usual output requirement in distributed algorithms. This result (Corollary 20) will be established through a connection with the concept of *low-congestion shortcuts*, which also implies other important results as well; e.g. for the SSSP problem (Corollary 21). We also present another simulation argument, leading to an accelerated algorithm for approximating the diameter. It should be stress that for this section we model the local network via the weaker CONGEST model.

4.1 Low-Congestion Shortcuts

Consider a graph $G = (V, E)$ under the CONGEST model, and a partition of V into k parts $P_1, \dots, P_k$ such that the induced graph $G[P_i]$ is connected. A recurring scenario in distributed algorithms consists of having to perform simultaneous aggregations in each part; this will be referred to as the *part-wise aggregation* problem. For example, an instance of this problem corresponds to determining the minimum-weight outgoing edge in the context of Boruvka's celebrated algorithm. A very insightful observation by Ghaffari and Haeupler [21] was to parameterize the performance of algorithms based on the complexity of the part-wise aggregation problem. For instance, if it admits a solution in Q rounds, under any collection of parts, we can compute an MST in $\mathcal{O}(Q \log n)$ rounds via Boruvka's algorithm. Now although in general graphs $Q = \mathcal{O}(\sqrt{n} + D)$ rounds, with the bound being existential tight for certain topologies, a key insight of Ghaffari and Haeupler [21] is that special classes of graphs allow for accelerated algorithms via *shortcuts*; most notably, for planar graphs they showed that the part-wise aggregation problem can be solved in $\widetilde{\mathcal{O}}(D)$ rounds of CONGEST, bypassing the notorious $\Omega(\sqrt{n})$ rounds for "global" problems under general graphs.

In the hybrid model this connection is particularly useful since the NCC model enables very fast algorithms for solving the part-wise aggregation problem:

▶ **Lemma 17** ([5]). *The part-wise aggregation problem admits a solution with high probability in $\mathcal{O}(\log n)$ rounds in* NCC*.*

As a result, we can directly derive a near-optimal algorithm for the minimum spanning tree problem through an implementation based on Boruvka's algorithm:

▶ **Corollary 18.** *There exists a distributed algorithm which computes with high probability an MST in $\mathcal{O}(\log^2 n)$ rounds of* CONGEST + NCC*.*

It should be noted that a deterministic $\mathcal{O}(\log^2 n)$ algorithm in CONGEST + NCC for the MST problem was developed in [18] with very different techniques. More importantly, Ghaffari and Haeupler [21] managed to establish the following:

▶ **Theorem 19** ([21]). *If we can solve the part-wise aggregation problem in Q rounds of* CONGEST*, there exists an $\widetilde{\mathcal{O}}(Q \operatorname{poly}(1/\epsilon))$ distributed algorithm for computing with high probability a $(1+\epsilon)$-approximation of the minimum cut, for any sufficiently small $\epsilon > 0$.*

We should remark that in [21] the authors establish this result only for planar graphs, but their argument can be directly extended in the form of this theorem. As a result, if we use Lemma 17 we arrive at the following conclusion:

▶ **Corollary 20.** *Consider any n-node weighted graph. There exists an $\mathcal{O}(\operatorname{polylog}(n))$-round algorithm in* CONGEST + NCC *for computing with high probability a $(1+\epsilon)$-approximation of Min-Cut, for any sufficiently small constant $\epsilon > 0$.*

In terms of *exact* Min-Cut, one can obtain an $\widetilde{\mathcal{O}}(\sqrt{n})$-round algorithm in CONGEST + NCC by simulating the recent algorithm due to Dory et al. [14], which requires $\widetilde{\mathcal{O}}(D + \sqrt{n})$ rounds of CONGEST; an analogous simulation argument is employed in the next subsection, so we omit the proof here. However, this leaves a substantial gap between exact and approximate Min-Cut. Moreover, analogous results can be established for computing approximate shortest paths by virtue of a result by Haeupler and Li [26]:

▶ **Corollary 21.** *Consider any n-node weighted graph. There exists with high probability a $\operatorname{polylog}(n)$-approximate algorithm for the single-source shortest paths problem which runs in $\widetilde{\mathcal{O}}(n^\epsilon)$ rounds in* CONGEST + NCC*, for any constant $\epsilon > 0$.*

We remark that Haeupler and Li [26] actually provide a more general result, but we state this special case for the sake of simplicity. Of course, there are other applications as well, as we have certainly not exhausted the literature. Overall, this connection illustrates another very concrete motivation of low-congestion shortcuts.

4.2 Diameter

We also provide another notable simulation argument. In particular, the main idea is to augment the local topology with a limited number of "global" edges so that the resulting graph has a small diameter, and at the same time the solution to the underlying problem remains invariant.

▶ **Proposition 22.** *There exists a distributed algorithm in* CONGEST + NCC *which determines a $3/2$-approximation of the diameter in $\mathcal{O}(\sqrt{n \log n})$ rounds with high probability.*

5 Distance Computations

5.1 Lower Bound for the Radius

In this subsection we show a lower bound of $\widetilde{\Omega}(n^{1/3})$ rounds for computing the radius – the smallest eccentricity of the graph – in the HYBRID model, even for unweighted graphs. We commence by constructing a suitable "gadget" in the BCC model, and then we will massage it appropriately to establish a guarantee for HYBRID as well. Our construction is inspired by that in [1] which established a sharp lower bound for sparse networks in CONGEST. First, consider a set of nodes $U \cup V \cup V' \cup U'$, and we let $U = \{u_0, u_1, \dots, u_{k-1}\}, V = \{v_0, v_1, \dots, v_{k-1}\}, V' = \{v'_0, v'_1, \dots, v'_{k-1}\}$, and $U' = \{u'_0, u'_1, \dots, u'_{k-1}\}$. We also incorporate edges of the form $\{v_i, v'_i\}$ for all $i \in \{0, 1, \dots, k-1\} = [k]^*$. An important additional ingredient is the *bit-gadget* [1], which works as follows: every node in U and U' will inherit some edge-connections based on the binary representation of their index; the role of this component will become clear as we proceed with the construction. Formally, consider a (different) set of nodes F, T, F', T', and let $F = \{f_0, f_1, \dots, f_{l-1}\}, T = \{t_0, t_1, \dots, t_{l-1}\}, F' = \{f'_0, f'_1, \dots, f'_{l-1}\}$, and $T' = \{t'_0, t'_1, \dots, t'_{l-1}\}$, where $l = \lceil \log k \rceil$. Now consider a node $u_i \in U$, and let $i = \overline{b_{l-1} \dots b_1 b_0}$ be the binary representation of its index; for every $j \in [l]^*$ we add the edge $\{u_i, f_j\}$ if $b_j = 0$; otherwise, we add the edge $\{u_i, t_j\}$. This process is also repeated for the nodes in U' (with respect to the sets F' and T'). Next, we add the edges $\{f_j, t_j\}$ and $\{f'_j, t'_j\}$ for all $j \in [l]^*$, while a critical element of the construction is the set of edges $\{\{f_j, t'_j\} : j \in [l]^*\} \cup \{\{t_j, f'_j\} : j \in [l]^*\}$. We also incorporate in the graph two nodes w, w' such that w is connected to all the nodes in U and V, and w' is connected to all the nodes in U' and V'. Finally, we add three nodes z_0, z_1, z_2, as well as the set of edges $\{\{z_0, z_1\}\} \cup \{\{z_1, z_2\}\} \cup \{\{z_0, u_i\} : i \in [k]^*\}$.

Having constructed this base graph the next step is to encode the input of Alice and Bob as edges on the induced graph. Specifically, let $x \in \{0,1\}^{k^2}$ and $y \in \{0,1\}^{k^2}$ represent the input strings of Alice and Bob respectively. We let $x_{i,j} = 1 \iff \{u_i, v_j\} \in E$, and $y_{i,j} = 1 \iff \{v'_j, u'_i\} \in E$. We denote the induced graph with $G_k^{x,y}$; an example of our construction is illustrated in Figure 2.

▷ Claim 23. For every node u in $G_k^{x,y}$ besides the nodes in U it follows that $\text{ecc}(u) \geq 4$.

▷ Claim 24. The radius R of $G_k^{x,y}$ is 3 if $x \cap y \neq \emptyset$; otherwise, $R = 4$.

▶ **Theorem 25.** *For any $\epsilon \in (0, 1/3]$, determining a $(4/3 - \epsilon)$-approximation for the radius of an unweighted graph with probability $2/3$ requires $\widetilde{\Omega}(n)$ rounds of* BCC.

Next, we will show how to adapt this construction for the HYBRID model. Specifically, if $\ell \in \mathbb{N}$ is some parameter, we introduce the following modifications: instead of connecting the corresponding nodes in V with V', F with T', and F' with T directly via edges, we will connect them via paths of length ℓ edges; moreover, we create the path $z_0 \to z_1 \to \dots \to z_{\ell+1}$ (in place of $z_0 \to z_1 \to z_2$). As before, the players' inputs $x, y \in \{0,1\}^{k^2}$ shall be encoded as edges between U with V, and V' with U' for Alice and Bob respectively. Let $G_{k,\ell}^{x,y}$ be the induced graph; the following claim admits an analogous proof to Claim 24:

▷ Claim 26. The radius R of $G_{k,\ell}^{x,y}$ is $\ell + 2$ if $x \cap y \neq \emptyset$; otherwise, $R \geq \ell + 3$.

Consequently, we are ready to state the implied lower bound in the HYBRID model for unweighted graphs.

▶ **Theorem 27.** *Determining the radius of an unweighted graph with probability $2/3$ requires $\widetilde{\Omega}(n^{1/3})$ rounds of* HYBRID.

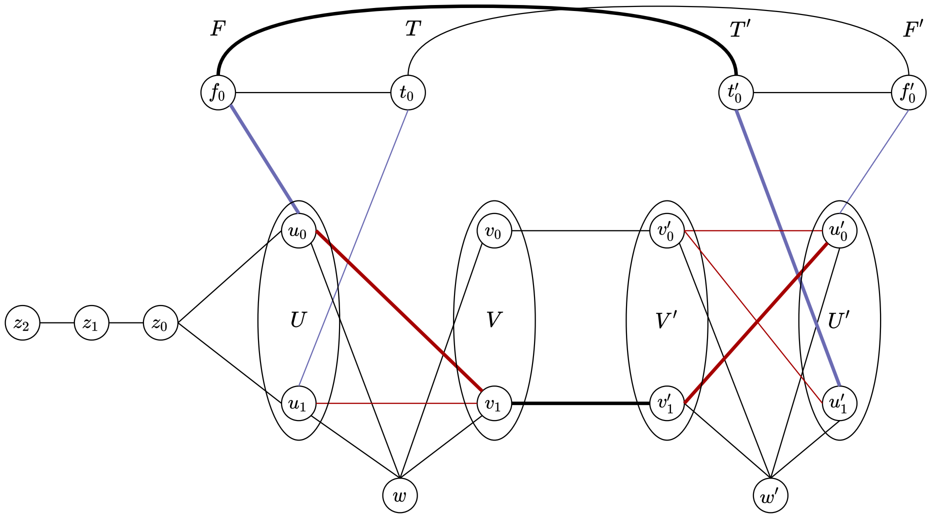


Figure 2 An example of our construction for the radius. The red edges correspond to the players' inputs, while the blue edges map nodes of U and U' to their *bit-gadget*. Observe that $d(u_0, u'_0) = 3$ (we have highlighted the corresponding path in the figure) as $\{u_0, v_1\} \in E$ and $\{v'_1, u'_0\} \in E$, implying that $x \cap y \neq \emptyset$. We have also highlighted the path of length 3 from u_0 to u'_1 through the bit-gadget.

Weighted Graphs

Next, we further modify our construction in order to obtain stronger lower bounds for weighted graphs. Specifically, if W represents the maximum-weight edge, we endow every edge of the graph $G^{x,y}_{k,\ell}$ with weight W, with the following exceptions: (i) all the edges belonging in paths connecting V to V'; (ii) all the edges belonging in paths connecting F to T' and T to F'; and (iii) all the edges belonging in the path $z_1 \to z_2 \to \cdots \to z_{\ell+1}$. We will represent the induced weighted graph as $G^{x,y}_{k,\ell,W}$.

▷ Claim 28. For every node u in $G^{x,y}_{k,\ell,W}$ besides the nodes in U it follows that $\mathrm{ecc}(u) \geq \ell + 3W$.

▷ Claim 29. The radius R of $G^{x,y}_{k,\ell,W}$ is $\ell + 2W$ if $x \cap y \neq \emptyset$; otherwise, $R = \ell + 3W$.

Observe that for sufficiently large W this claim implies an asymptotically 3/2-gap depending on whether $x \cap y = \emptyset$. As a result, we are ready to establish the following theorem:

▶ **Theorem 30.** *For any $\epsilon \in (0, 1/2]$, determining a $(3/2 - \epsilon)$-approximation for the radius of a weighted graph with probability $2/3$ requires $\widetilde{\Omega}(n^{1/3})$ rounds of* HYBRID*, assuming that $W = \omega(n^{1/3})$*

5.2 Diameter

Finally, we employ the machinery developed in [9] for solving in parallel multiple single-source shortest paths problems in HYBRID in order to derive an efficient implementation of the sequential algorithm of Roditty and Vassilevska W. [48] in HYBRID. This leads to an improved approximation algorithm for the diameter under graphs with small degrees:

▶ **Proposition 31.** *For any weighted graph G with max-degree $\Delta = \mathcal{O}(\operatorname{polylog} n)$ we can determine a 3/2-approximation of the diameter in $\widetilde{\mathcal{O}}(n^{1/3})$ rounds of* HYBRID*.*

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016*, volume 9888 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2016. doi:10.1007/978-3-662-53426-7_3.

2 Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. Subcubic equivalences between graph centrality problems, APSP and diameter. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 1681–1697. SIAM, 2015. doi:10.1137/1.9781611973730.112.

3 Ioannis Anagnostides and Themis Gouleakis. Deterministic distributed algorithms and lower bounds in the hybrid model, 2021. arXiv:2108.01740.

4 Sanjeev Arora, Satish Rao, and Umesh V. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2):5:1–5:37, 2009. doi:10.1145/1502793.1502794.

5 John Augustine, Mohsen Ghaffari, Robert Gmyr, Kristian Hinnenthal, Christian Scheideler, Fabian Kuhn, and Jason Li. Distributed computation in node-capacitated networks. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019*, pages 69–79. ACM, 2019. doi:10.1145/3323165.3323195.

6 John Augustine, Kristian Hinnenthal, Fabian Kuhn, Christian Scheideler, and Philipp Schneider. Shortest paths in a hybrid network model. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1280–1299. SIAM, 2020. doi:10.1137/1.9781611975994.78.

7 Joshua D. Batson, Daniel A. Spielman, Nikhil Srivastava, and Shang-Hua Teng. Spectral sparsification of graphs: theory and algorithms. *Commun. ACM*, 56(8):87–94, 2013. doi:10.1145/2492007.2492029.

8 András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, 1996*, pages 47–55. ACM, 1996. doi:10.1145/237814.237827.

9 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. Distance computations in the hybrid network model via oracle simulations, 2020. arXiv:2010.13831.

10 Keren Censor-Hillel, Dean Leitersdorf, and Volodymyr Polosukhin. On sparsity awareness in distributed computations. In Kunal Agrawal and Yossi Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, 2021*, pages 151–161. ACM, 2021. doi:10.1145/3409964.3461798.

11 Tao Chen, Xiaofeng Gao, and Guihai Chen. The features, hardware, and architectures of data center networks: A survey. *Journal of Parallel and Distributed Computing*, 96:45–74, 2016. doi:10.1016/j.jpdc.2016.05.009.

12 Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 273–282, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993636.1993674.

13 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing*, STOC '11, page 363–372. Association for Computing Machinery, 2011. doi:10.1145/1993636.1993686.

14 Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, 2021*, pages 1144–1153. ACM, 2021. doi:10.1145/3406325.3451020.

15 Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14*, pages 367–376. ACM, 2014. doi:10.1145/2611462.2611493.

16 Michael Elkin and Shaked Matar. Ultra-sparse near-additive emulators. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, 2021*, pages 235–246. ACM, 2021. doi:10.1145/3465084.3467926.

17 Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 339–350, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1851182.1851223.

18 Michael Feldmann, Kristian Hinnenthal, and Christian Scheideler. Fast hybrid network algorithms for shortest paths in sparse graphs, 2020. arXiv:2007.01191.

19 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, page 1150–1162. Society for Industrial and Applied Mathematics, 2012. doi:10.1137/1.9781611973099.91.

20 Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998. doi:10.1137/S0097539794261118.

21 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 202–219. SIAM, 2016. doi:10.1137/1.9781611974331.ch16.

22 Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In Yehuda Afek, editor, *Distributed Computing*, pages 1–15, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-41527-2_1.

23 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018*, volume 121 of *LIPIcs*, pages 29:1–29:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.29.

24 Mohsen Ghaffari and Krzysztof Nowicki. Congested clique algorithms for the minimum cut problem. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 357–366, New York, NY, USA, 2018. Association for Computing Machinery. URL: https://dl.acm.org/citation.cfm?id=3212750.

25 Thorsten Götte, Kristian Hinnenthal, Christian Scheideler, and Julian Werthmann. Time-optimal construction of overlay networks, 2020. arXiv:2009.03987.

26 Bernhard Haeupler and Jason Li. Faster distributed shortest path approximations via shortcuts. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018*, volume 121 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.33.

27 Johan Håstad and Avi Wigderson. The randomized communication complexity of set disjointness. *Theory Comput.*, 3(1):211–219, 2007. doi:10.4086/toc.2007.v003a011.

28 Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discret. Math.*, 5(4):545–557, 1992. doi:10.1137/0405044.

29 Udit Narayana Kar and Debarshi Kumar Sanyal. An overview of device-to-device communication in cellular networks. *ICT Express*, 4(4):203–208, 2018. doi:10.1016/j.icte.2017.08.002.

30 David R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, page 21–30, USA, 1993. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=313559.313605.

31 P. Klein, C. Stein, and É. Tardos. Leighton-rao might be practical: Faster approximation algorithms for concurrent flow with uniform capacities. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 310–321. Association for Computing Machinery, 1990. doi:10.1145/100216.100257.

32 Ioannis Koutis. Simple parallel and distributed algorithms for spectral graph sparsification, 2014. arXiv:1402.3851.

33 Fabian Kuhn and Philipp Schneider. Computing shortest paths and diameter in the hybrid network model. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 109–118. Association for Computing Machinery, 2020. doi:10.1145/3382734.3405719.

34 Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In Panagiota Fatourou and Gadi Taubenfeld, editors, *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 42–50. ACM, 2013. doi:10.1145/2484239.2501983.

35 Christoph Lenzen. *Lectures notes on Theory of Distributed Systems.* , 2016. URL: https://www.mpi-inf.mpg.de/fileadmin/inf/d1/teaching/winter15/tods/ToDS.pdf.

36 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

37 P. Li, S. Guo, and I. Stojmenovic. A truthful double auction for device-to-device communications in cellular networks. *IEEE Journal on Selected Areas in Communications*, 34(1):71–81, 2016. doi:10.1109/JSAC.2015.2452587.

38 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

39 Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. MST construction in O(log log n) communication rounds. In Arnold L. Rosenberg and Friedhelm Meyer auf der Heide, editors, *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 94–100. ACM, 2003. doi:10.1145/777412.777428.

40 Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web – analyzed on different aggregation levels. *The Journal of Web Science*, 1(1):33–47, 2015. doi:10.1561/106.00000003.

41 A. Murkaz, R. Hussain, S. F. Hasan, M. Y. Chung, B. . Seet, P. H. J. Chong, S. T. Shah, and S. A. Malik. Architecture and protocols for inter-cell device-to-device communication in 5G networks. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 489–492, 2016. doi:10.1109/DASC-PICom-DataCom-CyberSciTec.2016.95.

42 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In Fabian Kuhn, editor, *Distributed Computing*, pages 439–453, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-662-45174-8_30.

43 Noam Nisan and Ilya Segal. The communication requirements of efficient allocations and supporting prices. *Journal of Economic Theory*, 129:192–224, 2006. doi:10.1016/j.jet.2004.10.007.

44 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, USA, 2000.

45 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, page 253, USA, 1999. IEEE Computer Society. doi:10.1109/SFFCS.1999.814597.

46 David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(4):740–747, 1989. doi:10.1137/0218050.

47 Ran Raz and Avi Wigderson. Monotone circuits for matching require linear depth. *J. ACM*, 39(3):736–744, 1992. doi:10.1145/146637.146684.

48 Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13*, pages 515–524. ACM, 2013. doi:10.1145/2488608.2488673.

49 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 350–363. ACM, 2020. doi:10.1145/3357713.3384298.

50 C. Wang, F. Haider, X. Gao, X. You, Y. Yang, D. Yuan, H. M. Aggoune, H. Haas, S. Fletcher, and E. Hepsaydir. Cellular architecture and key technologies for 5G wireless communication networks. *IEEE Communications Magazine*, 52(2):122–130, 2014. doi:10.1109/MCOM.2014.6736752.

51 Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T.S. Eugene Ng, Michael Kozuch, and Michael Ryan. C-through: Part-time optics in data centers. *SIGCOMM Comput. Commun. Rev.*, 40(4):327–338, August 2010. doi:10.1145/1851182.1851222.

52 Andrew Chi-Chih Yao. Some complexity questions related to distributive computing(preliminary report). In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, page 209–213, New York, NY, USA, 1979. Association for Computing Machinery. doi:10.1145/800135.804414.

Ruling Sets in Random Order and Adversarial Streams

Sepehr Assadi ✉
Department of Computer Science, Rutgers University, Piscataway, NJ, USA

Aditi Dudeja ✉
Department of Computer Science, Rutgers University, Piscataway, NJ, USA

Abstract

The goal of this paper is to understand the complexity of a key symmetry breaking problem, namely the (α, β)-ruling set problem in the graph streaming model. Given a graph $G = (V, E)$, an (α, β)-ruling set is a subset $I \subseteq V$ such that the distance between any two vertices in I is at least α and the distance between a vertex in V and the closest vertex in I is at most β. This is a fundamental problem in distributed computing where it finds applications as a useful subroutine for other problems such as maximal matching, distributed colouring, or shortest paths. Additionally, it is a generalization of MIS, which is a $(2, 1)$-ruling set.

Our main results are two algorithms for $(2, 2)$-ruling sets:

1. In adversarial streams, where the order in which edges arrive is arbitrary, we give an algorithm with $\tilde{O}(n^{4/3})$ space, improving upon the best known algorithm due to Konrad et al. [DISC 2019], with space $\tilde{O}(n^{3/2})$.
2. In random-order streams, where the edges arrive in a random order, we give a semi-streaming algorithm, that is an algorithm that takes $\tilde{O}(n)$ space.

Finally, we present new algorithms and lower bounds for (α, β)-ruling sets for other values of α and β. Our algorithms improve and generalize the previous work of Konrad et al. [DISC 2019] for $(2, \beta)$-ruling sets, while our lower bound establishes the impossibility of obtaining any non-trivial streaming algorithm for $(\alpha, \alpha - 1)$-ruling sets for all even $\alpha > 2$.

2012 ACM Subject Classification Theory of computation → Sketching and sampling; Theory of computation → Lower bounds and information complexity; Theory of computation → Random order and robust communication complexity

Keywords and phrases Symmetry breaking, Ruling sets, Lower bounds, Communication Complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.6

Funding *Sepehr Assadi*: Research supported in part by a NSF CAREER Grant CCF-2047061, and a gift from Google Research.
Aditi Dudeja: This work was done while funded by NSF Award 1942010.

Acknowledgements We thank anonymous reviewers for their insightful feedback.

1 Introduction

The goal of this paper is to understand the complexity of *symmetry breaking problems*, specifically the (α, β)-ruling set problem, which is defined as follows.

▶ **Definition 1.** *Given a graph $G = (V, E)$, an (α, β)-ruling set is a subset $I \subseteq V$ such that the distance between any two vertices in I is at least α and the distance between a vertex in V and the closest vertex in I is at most β.*

Ruling sets are a generalization of *maximal independent sets*: MIS is a $(2, 1)$-ruling set. Ruling sets have been well-studied in numerous distributed setting such as the CONGEST and LOCAL models, and the k-machine model (see e.g. [7, 8, 10, 17, 18, 24, 31]). Moreover,

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 6; pp. 6:1–6:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ruling sets are used as a subroutine in algorithms solving other interesting graph problems such as distributed coloring [12], maximal matching [8], maximal independent set [18], or shortest paths [19]. In many applications where they are sufficient and easier to compute, ruling sets can replace MIS computation. (see e.g. [8, 27]).

We study this problem in the graph streaming model, which is motivated by the fact that modern data sets are too large to fit in a computer's random access memory. A streaming algorithm processes its data sequentially, in one or a small number of passes. It has a memory sublinear in the size of input. The distributed and graph streaming models have a lot in common, with techniques used in one model often finding applications in the other. For example L_0-sampling [20] and graph sketches [3] which were first used in the context of streaming algorithms, now find application in several distributed settings too: [30] use linear sketches to obtain algorithms for MST in the k-machine model, and [21] use them in the Congested Clique model to give an optimal algorithm for the same problem. There are numerous examples of tools from communication and information complexity being used to establish lower bounds for streaming as well as distributed computing problems (see, e.g. [1] and the references therein).

Many of the classical symmetry breaking problems have been studied in the semi-streaming model. For example, [6] and [13] studied the MIS problem in the semi-streaming setting and obtained an $\Omega(n^2)$ lower bound on space for one pass streaming algorithms. If we increase the number of passes, then correlation clustering of [2] enables us to get a semi-streaming algorithm in $O(\log\log n)$ passes. Similar to the case of distributed computing where the problem of computing a $(\Delta{+}1)$-colouring is easier than computing an MIS, [6] also gave a one-pass semi streaming algorithm for $\Delta+1$ coloring. [24] studied the problem of $(2,\beta)$-ruling sets in the semi-streaming setting, establishing that for $\beta \geq 2$, the problem of computing a $(2,\beta)$-ruling set is strictly easier than computing an MIS. We extend this line of study of symmetry breaking problems by considering random order streams, and giving improved algorithms and lower bounds for adversarial streams for various values of α and β.

Our Results. Our main result is a semi-streaming algorithm for $(2,2)$-ruling sets and thus $(2,\beta)$-ruling sets for all $\beta \geq 2$ in random-order streams. The random order model is motivated by the fact that it gives rise to a natural notion of *average-case analysis*, which explains why some streaming problems may have strong lower bounds, while being efficiently solvable in practice (see [28]). As a result, several problems such as matching (see [23, 16, 22, 15, 5, 4, 9]), connectivity [11] or properties of bounded degree graphs [29] have been studied extensively in this model. With this, we state our first result.

▶ **Result 1.** *There is an $\tilde{O}(n)$-space streaming algorithm that computes a $(2,2)$-ruling set (and therefore $(2,\beta)$-ruling set for $\beta \geq 2$) of any graph with high probability when the edges arrive in a random order. This also gives a semi-streaming algorithm for (α,β)-ruling sets for $\beta \geq \alpha \geq 2$ when the edges arrive in a random order.*

Result 1 is the first example of a semi-streaming algorithm for any $(2,\beta)$-ruling set for $\beta = O(1)$ (although we make the assumption that the edges arrive in a random order). This bound is optimal since $\Omega(n)$ space is needed to output the solution. While it is known that MIS requires $\Omega(n^2)$ space when the edges arrive in the stream in an arbitrary order, no such barriers are known for random-order stream. Thus resolving the complexity of MIS in this model remains an interesting open question.

Our second contribution is improving the $\tilde{O}(n^{1+\frac{1}{2^{\beta}-1}})$-space algorithm of [24] for $(2,\beta)$-ruling sets when edges arrive in the stream in an arbitrary order.

▶ **Result 2.** *There is an* $\tilde{O}(\beta \cdot n^{1+\frac{1}{2^\beta-1}})$ *space streaming algorithm that computes a* $(2,\beta)$*-ruling set of any graph with high probability when the edges arrive in an arbitrary order. This also gives a streaming algorithm that computes an* (α,β)*-ruling set (for* $\beta \geq \alpha \geq 2$*) with high probability using* $\tilde{O}(n^{1+\frac{1}{2^{\beta-\alpha+2}-1}})$ *space when the edges arrive in an arbitrary order*

The improvement in Result 2 is the most significant for $\alpha = \beta = 2$, where we get a bound of $\tilde{O}(n^{4/3})$, improving upon the previously known best bound of $\tilde{O}(n^{3/2})$. Our final result is a lower bound for $(\alpha, \alpha-1)$-ruling sets for all even $\alpha > 2$.

▶ **Result 3.** *Any randomized constant error one-pass streaming algorithm in the adversarial order model that computes a* $(\alpha,\alpha-1)$*-ruling set for any even* $\alpha > 2$ *requires* $\Omega(n^2/\alpha^2)$ *space.*

Landscape of (α,β)-ruling sets. The algorithm of Result 2 gives non-trivial streaming algorithms (with space complexity much smaller than input size), for all $\beta \geq \alpha \geq 2$. On the other hand, Result 3 establishes that the problem of computing (α,β)-ruling sets when $\beta = \alpha - 1$ for even $\alpha > 2$, does not admit non-trivial streaming algorithms. When $\beta < \alpha - 1$, an (α,β)-ruling sets may not even exist. Thus, our results give a relatively complete picture of which setting of parameters α and β the problem of computing (α,β)-ruling sets admits non-trivial streaming algorithms.

2 Preliminaries

From now on, we use β-ruling sets to denote $(2,\beta)$-ruling sets. For a graph G we use n to denote $|V(G)|$ and m to denote $|E(G)|$. For a subgraph K of G and a vertex $v \in K$, we define $E_K(v)$ to be the set of edges incident on v in K, $\deg_K(v)$ to be the number of neighbours of v in K and $N_K(v)$ to be the neighbours of v in K. For $u, v \in V(G)$, let $\text{dist}(u,v)$ denote the length of the shortest path from u to v.

Concentration Results. In our proofs, we will use negatively associated random variables.

▶ **Definition 2.** *We say a collection of random variables* $\{X_1, X_2, \cdots, X_k\}$ *are* negatively associated *if for any two disjoint index sets* $I, J \subseteq [n]$ *and two functions* f, g *both monotone increasing or both monotone decreasing, it holds*

$$\mathbb{E}\left[f(X_i : i \in I) \cdot g(X_j : j \in J)\right] \leq \mathbb{E}\left[f(X_i : i \in I)\right] \cdot \mathbb{E}\left[g(X_j : j \in J)\right]$$

We will apply our concentration bounds to randomly chosen subsets of vertices of a fixed size, or to segments of stream. Since the stream is random-ordered, both of these situations can be thought of as sampling without replacement. Thus, we can apply Chernoff for negatively associated random variables (see for example [32]).

▶ **Proposition 3** ([14]). [Chernoff Bound] *Let* $X_1, \cdots, X_k$ *be negatively associated* 0-1 *random variables. Let* $X = \sum_{i=1}^k X_i$*. Let* $\mu = \mathbb{E}\left[\sum_{i=1}^k X_i\right]$ *and let* $\mu_{\min} \leq \mu \leq \mu_{\max}$*. Then, for all* $\delta \in (0,1)$*, we have:*

$$\Pr\left(X \geq (1+\delta)\mu_{\max}\right) \leq \exp\left(-\delta^2\mu_{\max}/3\right) \text{ and } \Pr\left(X \leq (1-\delta)\mu_{\min}\right) \quad \leq \exp\left(-\delta^2\mu_{\min}/2\right).$$

Communication Complexity. In order to prove lower bounds on the space complexity, we will reduce Index_t to our problem, which we formally define.

▶ **Definition 4.** *In the two-party communication problem* Index_t*, Alice holds a* t*-bit string* $X \in \{0,1\}^t$ *and Bob holds an index* $\sigma \in [t]$*. Alice sends a single message to Bob, who upon receipt outputs* X_σ*.*

We shall use the following well-known distribution for Index_t.

Distribution 1 Distribution $\mathcal{D}_{\mathsf{Ind}}$.

```
1 Pick $X \in \{0,1\}^t$ uniformly at random.
2 Pick $\sigma \in [t]$ uniformly at random.
```

If an instance of Index_t is drawn from $\mathcal{D}_{\mathsf{Ind}}$, then we have the following bound on the message complexity of any communication protocol that solves it (see [26]).

▶ **Proposition 5.** *For* $0 < \delta < 1/2$*, any* δ*-error protocol for* Index_t *over* $\mathcal{D}_{\mathsf{Ind}}$*, ccmmunicates* $\Omega(t)$ *bits.*

3 2-Ruling Sets in Random-Order Streams

In this section, we state our main contribution: We give a simple streaming algorithm that computes a 2-ruling set of a given graph in **random-order stream** in $\tilde{O}(n)$ space. We now state our main theorem formally.

▶ **Theorem 6.** *There is an* $\tilde{O}(n)$*-space streaming algorithm that computes a* 2*-ruling set of any graph* G *with high probability when the edges of* G *arrive in a random order in the stream.*

In Section 3.1 we will show a structural theorem that we will use to compute 2-ruling sets in random order stream. In the subsequent section, we will state our algorithm and argue its correctness and space complexity.

3.1 Peeling Decomposition

In order to compute 2-Ruling Sets in random order streams, we will use a structural decomposition of the graph G. We formally state this decomposition. A similar decomposition was used first by [25] and [10] to compute β-ruling sets in the distributed setting, and then subsequently by [24] to get streaming algorithms for β-ruling sets. Since this decomposition was given in a different language in the previous works, we re-state it here for completeness. Our main contribution is the observation that this decomposition can be computed in the random-order setting with high probability and using a small amount of space.

▶ **Definition 7.** *Let* $G = (V, E)$ *be a graph, let* $r = \log n - \log\log n - 7$ *and let* $(d_0, d_1, \cdots, d_r)$ *be a sequence of integers such that* $d_i = n/2^i$ *for* $i \in [r]$*. We define the* ***Peeling Decomposition*** $(G_0, G_1, \cdots, G_r)$ *of* G *as follows.*

1. *Each* G_i *consists of vertices* V_i *and edges* E_i*.*
2. *We let* $G_0 = G$*, and* $G_i \subseteq G_{i-1}$ *for all* $i \geq 0$*.*
3. *For* $i \geq 1$*, we define* $V_i = \left\{v \in V_{i-1} \mid deg_{G_{i-1}}(v) \leq d_i\right\}$*,* $G_i = G[V_i]$ *and* $E_i = E(G_i)$*.*

Depending on the decomposition, with each vertex $v \in V$, we associate a level.

▶ **Definition 8.** *For every vertex* v*, we define the level of a vertex to be the unique index* l *such that* $v \in V_l$ *but* $v \notin V_{l+1}$*. If* $v \in V_r$ *then the level of* v *is* r*.*

We now describe a process to create an induced subgraph H of G whose MIS with high probability will be a 2-ruling set for G. Moreover, the total number of edges in H will be small.

Process 2 A Process for Sampling a 2-Ruling Set.

```
Input: Graph $G$, $(d_0, d_1, \cdots, d_r)$, where $r = \log n - \log\log n - 7$, $d_i = n/2^i$ and
       peeling decomposition $(G_0, \cdots, G_r)$.
For each $i \in \{1, 2, \cdots, r-1\}$, we sample $S_i$ of size $\frac{10 \cdot |V_i| \cdot \log n}{d_{i+1}}$ from $V_i$ uniformly at
  random.
Let $H$ be the induced subgraph $G[\cup_{i=0}^{r-1} S_i \cup V_r]$.
Return $\mathcal{M}$, an arbitrary MIS of $H$.
```

We now prove the following guarantees on H and $\mathcal{M}$.

▶ **Lemma 9.** *The set $\mathcal{M}$ returned by Process 2 is a 2-ruling set of G with high probability.*

Proof. We first note that since H is an induced subgraph of G, $\mathcal{M}$ is an independent set in G. We want to show that for all $v \notin \mathcal{M}$, there is a vertex $u \in \mathcal{M}$ such that u is at most distance two from v.

Towards this, we first consider a vertex $v \in V_r$. Observe that since $V_r \subseteq V(H)$, this implies that either $v \in \mathcal{M}$ or there is a vertex $u \in \mathcal{M}$ such that $\text{dist}(v, u) = 1$. We now prove the claim for vertices on level $i < r$. Consider a vertex $v \in V_{i-1} \setminus V_i$. By definition, we know that $\deg_{G_{i-1}}(v) \geq d_i$. For a vertex $w \in N_{G_{i-1}}(v)$, we define X_w to be the indicator random variable that takes value 1 if $w \in S_{i-1}$ and value 0 if $w \notin S_{i-1}$. Since S_{i-1} is sampled uniformly at random from V_{i-1} and $|S_{i-1}| = \frac{10 \cdot |V_{i-1}| \cdot \log n}{d_i}$, we have the following bound for a vertex $w \in N_{G_{i-1}}(v)$.

$$\Pr(X_w = 1) = \frac{|S_{i-1}|}{|V_{i-1}|} = \frac{10 \cdot \log n}{d_i}$$

Note that the above bound is at most 1, since $d_r \geq 100 \cdot \log n$, by our choice of r and d_i's. Observe that the random variables $\left\{X_w \mid w \in N_{G_{i-1}}(v)\right\}$ are negatively associated (since they correspond to sampling without replacement). This gives us the following bounds:

$$\Pr\left(\bigwedge_{w \in N_{G_{i-1}}(v)} X_w = 0\right) \leq \prod_{w \in N_{G_{i-1}}(v)} \Pr(X_w = 0) = \left(1 - \frac{10 \cdot \log n}{d_i}\right)^{\deg_{G_{i-1}}(v)}$$
$$\leq \left(1 - \frac{10 \cdot \log n}{d_i}\right)^{d_i} = O\left(\frac{1}{n^{10}}\right).$$
$$(\text{Since } \deg_{G_{i-1}}(v) \geq d_i)$$

Taking a union bound over all vertices we conclude that with probability $1 - o(1)$ for $i \in [r]$, for all $v \in V_{i-1} \setminus V_i$, there is a vertex $w \in N_{G_{i-1}}(v)$ such that $w \in S_{i-1}$. This proves our claim. ◀

We now bound the total number of edges in H.

▶ **Lemma 10.** *The total number of edges in H, the induced subgraph of Process 2, is at most $O(\sum_{i=0}^{r-1} \frac{|V_i| \cdot d_i \cdot \log n}{d_{i+1}} + n \cdot d_r)$. By our choice of d_i's and r, this is at most $\tilde{O}(n)$.*

Proof. To bound the total number of edges in H, we give a charging argument. For every edge (u, v) we charge this edge to the endpoint with the lower level. We consider the following two cases.

1. **Total charge on $v \in V_r$.** We begin by observing, that a vertex $v \in V_r$ is charged an edge (u, v) then level of u is also r (in other words, $u \in V_r$). Since $\deg_{G_{r-1}}(v) \leq d_r$, by definition of V_r, and $V_r \subseteq V_{r-1}$, we conclude that $\deg_{G_r}(v) \leq d_r$. From this discussion, we know that the total charge on any $v \in V_r$ is at most d_r. Thus, the total charge on V_r is at most $|V_r| \cdot d_r$.
2. **Total charge on $v \in S_i$.** Consider a vertex v at level i, and suppose it is charged an edge (u, v). Let j be the level of u. By our charging scheme, $j \geq i$. Consequently, $u \in V_j \subseteq V_i \subseteq V_{i-1}$. Additionally, we know by definition of V_i that $\deg_{G_{i-1}}(v) \leq d_i$. By the above discussion, the total charge on v is at most d_i. Therefore, the total charge on S_i is at most $\frac{100 \cdot d_i \cdot |V_i| \cdot \log n}{d_{i+1}}$.

This proves our claim. ◀

3.2 The Algorithm

We briefly describe our algorithm. To give our algorithm, we would ideally like to sample the subgraph H as described. If we were able to achieve this, then at the end of the stream we could compute an MIS $\mathcal{M}$ of H, and using the Lemma 9, we would know this is a 2-ruling set of G with high probability. Moreover using Lemma 10, we would be able to conclude that we have a semi-streaming algorithm for this problem.

However, determining V_i even in random order streams seems impossible. Therefore, instead of computing V_i, we will compute sets $\tilde{V}_i$ and $\tilde{G}_i = G[\tilde{V}_i]$, which will have the following relaxed property: for all vertices $v \in \tilde{V}_i$, $\deg_{\tilde{G}_{i-1}}(v) \leq d_i$, and for all $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, $\deg_{\tilde{G}_{i-1}}(v) \geq d_i/2$. We can then show that the main property still holds: for $i \in \{1, \cdots, r\}$, suppose we sample $\tilde{S}_{i-1}$ of size $\frac{100 \cdot |\tilde{V}_{i-1}| \cdot \log n}{d_i}$ uniformly at random from $\tilde{V}_{i-1}$ then with high probability, for all $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, there is $u \in N_{\tilde{G}_{i-1}}(v)$ such that $u \in \tilde{S}_{i-1}$. Finally, if $H = \cup_{i=1}^{r-1} \tilde{S}_i \cup \tilde{V}_r$, then an argument similar to the one in Lemma 10 will give us a similar bound on $|E(H)|$.

We give some intuition about our algorithm. Our algorithm starts by first guessing $\tilde{V}_1$ described above as follows: given parameter d_1, we sample a set $\tilde{S}_0$ of size $\frac{10 \cdot n \cdot \log n}{d_1}$ uniformly at random from V. We determine the set $\tilde{V}_1$ by filtering out vertices that have a lot of edges to $\tilde{S}_0$. To do this we only look at a small portion of the stream (the first $\frac{100 \cdot m \cdot \log n}{d_1}$ edges), and remove vertices that have a lot of edges incident on them in this portion of the stream. We will argue via a Chernoff bound argument that $\tilde{V}_1$ has the above property with high probability: all vertices in $\tilde{V}_1$ have few edges to $\tilde{S}_0$, and all vertices in $V \setminus \tilde{V}_1$ have a lot of edges to $\tilde{S}_0$. For any $i \geq 2$, we repeat this process inductively: suppose we have $\tilde{V}_{i-1}$. We sample $\tilde{S}_{i-1}$ from it, and we obtain $\tilde{V}_i$ by looking at a small portion of the stream and removing vertices that have a lot of edges to $\tilde{S}_{i-1}$.

We formally describe our algorithm in Algorithm 3, and we now show its correctness and space complexity.

▷ Claim 11. Let $\tilde{V}_1, \cdots \tilde{V}_r$ be the sets computed by Algorithm 3. Then, with high probability the following facts hold for all $i \in [r]$.

1. For any $v \in \tilde{V}_i$, $\deg_{\tilde{G}_{i-1}}(v) \leq d_i$.
2. For any $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, $\deg_{\tilde{G}_{i-1}}(v) \geq d_i/2$.

Proof. Consider $v \in \tilde{V}_i$, we want to show $\deg_{\tilde{G}_{i-1}}(v) \leq d_i$. Recall how $\tilde{V}_i$ is created: we add $v \in \tilde{V}_{i-1}$ to $\tilde{V}_i$ if we see fewer than $70 \log n$ edges from $E_{\tilde{G}_{i-1}}(v)$ among the first $\frac{100 \cdot m \log n}{d_i}$ edges that have arrived in the stream. Suppose $\deg_{\tilde{G}_{i-1}}(v) > d_i$. Let Y_v denote the random variable $|E_{\tilde{G}_{i-1}}(v) \cap \mathsf{Stream}_{i-1}|$. Observe that Y_v is a sum of negatively associated 0-1 random

Algorithm 3 Computing a 2-Ruling Set in Random-Order Streams.

```
Input: Integers $r = \log n - \log\log n - 7$, $(d_0, d_1, \cdots, d_r)$, where $d_i = n/2^i$.
Phase 0:
  $\tilde{V}_1 \leftarrow V$
  Let $\tilde{S}_0$ be chosen uniformly at random from $V$, where $|\tilde{S}_0| = \frac{10 \cdot n \cdot \log n}{d_1}$
  Store the first $\frac{100 \cdot m \cdot \log n}{d_1}$ edges of $G$ that arrive in the stream (denoted by $\mathsf{Stream}_0$).
  For all vertices $v$ with $|E_G(v) \cap \mathsf{Stream}_0| > 70 \log n$, let $\tilde{V}_1 \leftarrow \tilde{V}_1 \setminus \{v\}$.
  Discard all edges $(u, v)$ such that either $u \notin \tilde{S}_0 \cup \tilde{V}_1$ or $v \notin \tilde{S}_0 \cup \tilde{V}_1$.
Phase $i$ for $i \in \{1, 2, \cdots, r-1\}$:
  Initialize $\tilde{V}_{i+1} \leftarrow \tilde{V}_i$.
  Sample $\tilde{S}_i$ uniformly at random from $\tilde{V}_i$, where $|\tilde{S}_i| = \frac{10 \cdot |\tilde{V}_i| \cdot \log n}{d_{i+1}}$.
  Let $\tilde{G}_i = G[\tilde{V}_i]$. Let $\tilde{H}_i = G[\cup_{j=0}^{i-1} \tilde{S}_j \cup \tilde{V}_i]$. Process the stream, storing only edges of $\tilde{H}_i$.
    After the first $\frac{100 \cdot m \cdot \log n}{d_{i+1}}$ edges of $G$ (denoted $\mathsf{Stream}_i$) have been seen, do the following:
    for all vertices $v \in \tilde{V}_i$ with $|E_{\tilde{G}_i}(v) \cap \mathsf{Stream}_i| > 70 \log n$, let $\tilde{V}_{i+1} \leftarrow \tilde{V}_{i+1} \setminus \{v\}$.
  Discard all edges $(u, v)$ with $u \notin \cup_{j=0}^{i} \tilde{S}_j \cup \tilde{V}_{i+1}$ or $v \notin \cup_{j=0}^{i} \tilde{S}_j \cup \tilde{V}_{i+1}$.
After Phase $r - 1$:
  Process the stream, while only retaining edges that have both endpoints in $\cup_{j=0}^{r-1} \tilde{S}_j \cup \tilde{V}_r$.
  At the end of the stream return $\mathcal{M} = \mathsf{MIS}(G[\cup_{j=0}^{r-1} \tilde{S}_j \cup \tilde{V}_r])$.
```

variables. We briefly explain why: for an edge $e \in E_{\tilde{G}_{i-1}}(v)$, define X_e to be the indicator random variable that takes value 1 when $e \in \mathsf{Stream}_{i-1}$ and 0 otherwise. Note that X_e are negatively associated since they correspond to sampling $\frac{100 \cdot m \cdot \log n}{d_i}$ edges without replacement. We additionally have the following bound for $e \in E_{\tilde{G}_{i-1}}(v)$.

$$\Pr\left(X_e = 1\right) = \left(\frac{1}{m}\right)\left(\frac{100 \cdot m \cdot \log n}{d_i}\right)$$

The above bound is less than 1 due to the fact that $r = \log n - \log\log n - 7$ and $d_r > 100 \log n$. This gives us the following bound on $\mathbb{E}\left[X_v\right]$:

$$\mathbb{E}\left[Y_v\right] = \left(\frac{\deg_{\tilde{G}_{i-1}}(v)}{m}\right)\left(\frac{100 \cdot m \cdot \log n}{d_i}\right) > \left(\frac{d_i}{m}\right)\left(\frac{100 \cdot m \cdot \log n}{d_i}\right) = 100 \log n$$

We now apply Proposition 3 with $\delta = 3/10$ and $\mu_{\min} = 100 \log n$, to get:

$$\Pr\left(Y_v \leq 70 \log n\right) \leq \exp\left(-(3/10)^2 (100 \log n)(1/2)\right) = O\left(\frac{1}{n^4}\right)$$

$$\Pr\left(Y_v > 70 \log n\right) = 1 - O\left(\frac{1}{n^4}\right)$$

This implies that with high probability, v would be omitted from $\tilde{V}_i$, which is a contradiction.

We now prove that with high probability, for all $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, $\deg_{\tilde{G}_{i-1}}(v) \geq d_i/2$. The proof strategy will be same as before. Assume that $\deg_{\tilde{G}_{i-1}}(v) < d_i/2$. For $e \in E_{\tilde{G}_{i-1}}(v)$, we let X_e be a random variable that takes value 1 if $e \in \mathsf{Stream}_{i-1}$ and 0 otherwise. Let $Y_v = |E_{\tilde{G}_{i-1}}(v) \cap \mathsf{Stream}_{i-1}|$. As discussed before, we know that Y_v is a sum of 0-1 negatively correlated random variables $\left\{X_e \mid e \in E_{\tilde{G}_{i-1}}(v)\right\}$.

$$\Pr(X_e = 1) = \left(\frac{1}{m}\right)\left(\frac{100 \cdot m \cdot \log n}{d_i}\right)$$

This gives us the following bound on $\mathbb{E}[Y_v]$.

$$\mathbb{E}[Y_v] = \left(\frac{\deg_{\tilde{G}_{i-1}}(v)}{m}\right)\left(\frac{100 \cdot m \cdot \log n}{d_i}\right) < \left(\frac{d_i}{2m}\right)\left(\frac{100 \cdot m \cdot \log n}{d_i}\right) = 50 \log n$$

We now apply Proposition 3 with $\delta = 2/5$ and $\mu_{\max} = 50 \log n$.

$$\Pr(Y_v \geq 70 \log n) \leq \exp(-(2/5)^2 (50 \log n)(1/3)) = O\left(\frac{1}{n^2}\right)$$

$$\Pr(Y_v < 70 \log n) > 1 - O\left(\frac{1}{n^2}\right).$$

This implies that with high probability, v would be included in $\tilde{V}_i$ which is a contradiction. Taking a union bound over all vertices, we conclude the proof of the lemma. ◁

▷ Claim 12. For $i \in [r]$, consider any $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, there is a neighbour of v in $\tilde{S}_{i-1}$ with high probability.

Proof. For the proof of this lemma, we condition on Claim 11. The set $\tilde{S}_{i-1}$ is sampled uniformly at random from $\tilde{V}_{i-1}$, and $|\tilde{S}_{i-1}| = \frac{10 \cdot |\tilde{V}_{i-1}| \cdot \log n}{d_i}$. Consider any $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, from Claim 11, we know that $\deg_{\tilde{G}_{i-1}}(v) \geq \frac{d_i}{2}$. For $w \in N_{\tilde{G}_{i-1}}(v)$, consider 0-1 random variable X_w, that takes value 1 if $w \in \tilde{S}_{i-1}$ and 0 otherwise. The random variables $\left\{X_w = 1 \mid w \in N_{\tilde{G}_{i-1}}(v)\right\}$ are negatively associated. Let Y_v denote $|N_{\tilde{G}_{i-1}}(v) \cap \tilde{S}_{i-1}|$. We know that Y_v is a sum of negatively associated 0-1 random variables. Having established this notation, for $w \in N_{\tilde{G}_{i-1}}(v)$, we have the following fact.

$$\Pr(X_w = 1) = \frac{|\tilde{S}_{i-1}|}{|\tilde{V}_{i-1}|} = \left(\frac{10 \cdot |\tilde{V}_{i-1}| \cdot \log n}{d_i}\right)\left(\frac{1}{|\tilde{V}_{i-1}|}\right) = \frac{10 \log n}{d_i}.$$

The above bound is at most 1, since $d_r > 100 \cdot \log n$ by our choice of r and d_i's. We now bound the probability that none of the vertices in $N_{\tilde{G}_{i-1}}(v)$ are included in $\tilde{S}_{i-1}$.

$$\begin{aligned}\Pr\left(\bigwedge_{w \in N_{\tilde{G}_{i-1}}(v)} \{X_w = 0\}\right) &\leq \prod_{w \in N_{\tilde{G}_{i-1}}(v)} \Pr(X_w = 0) = \left(1 - \frac{10 \cdot \log n}{d_i}\right)^{\deg_{\tilde{G}_{i-1}}(v)} \\ &\text{(Negatively associated r.v.)} \\ &\leq \left(1 - \frac{10 \cdot \log n}{d_i}\right)^{d_i/2} = O\left(\frac{1}{n^5}\right) \\ &(\deg_{\tilde{G}_{i-1}}(v) \geq d_i/2).\end{aligned}$$

Taking a union bound over all $v \in V$, we conclude that for all $i \in [r]$ and for all $v \in \tilde{V}_{i-1} \setminus \tilde{V}_i$, there is $w \in N_{\tilde{G}_{i-1}}(v)$ such that $w \in \tilde{S}_{i-1}$ with high probability. ◁

Correctness. To show correctness, note that Algorithm 3 retains all edges of $H = G\left[\cup_{j=0}^{r-1}\tilde{S}_j \cup \tilde{V}_r\right]$ till the end of the stream. This implies that the set of vertices $\mathcal{M}$ output by the algorithm, indeed forms an MIS in H. Since H is an induced subgraph of G, $\mathcal{M}$ is also an independent set in G. We know that all $v \in V(H)$ are 1-ruled. Additionally, from Claim 12 and Claim 11 we conclude that for all $i \in [r-1]$, for all $v \in \tilde{V}_i \setminus \tilde{V}_{i+1}$, there is a neighbour of v in $\tilde{S}_i$. Let u be this neighbour. Since $\mathcal{M}$ is an MIS, either $u \in \mathcal{M}$ or there is $x \in N_G(u)$ such that $x \in \mathcal{M}$. This implies that v is 2-ruled. We conclude that with probability at least $1 - o(1)$, all vertices in V are 2-ruled.

Space Complexity. We show that the space complexity of this algorithm. Towards this, we show that the total number of edges stored by the algorithm is at most $O(\sum_{i=0}^{r-1}\frac{|\tilde{V}_i|\cdot d_i\cdot\log n}{d_{i+1}} + n\cdot d_r) = \tilde{O}(n)$ since $r = \log n - \log\log n - 7$ and $d_i = {}^{n}/_{2^i}$.

▶ **Lemma 13.** *The total number of edges stored by Algorithm 3 is* $O(\sum_{i=0}^{r-1}\frac{150\cdot|\tilde{V}_i|\cdot d_i\cdot\log n}{d_{i+1}} + n\cdot d_r)$ *with high probability.*

Proof. We start with showing the total number of edges in the memory at the end of Phase 0. Observe that Phase 0 ends when we have seen first $\frac{100\cdot m\cdot\log n}{d_1}$ edges of G. Therefore, by the end of this phase we have at most $\frac{100\cdot m\cdot\log n}{d_1}$ edges in the memory, which is $\frac{100\cdot n^2\cdot\log n}{d_1}$, the first term in the sum given in the lemma. During Phase i, we only store edges of $\tilde{H}_i = G\left[\cup_{j=0}^{i-1}\tilde{S}_j \cup \tilde{V}_i\right]$. We will now bound the number of such edges.

As before, we will make a charging argument. Recall the definition of the level of a vertex in Definition 8. Consider an edge $(u, v) \in \tilde{H}_i$ that appears in Stream_i. We charge (u, v) to the vertex with the lower level, breaking ties arbitrarily. Suppose v is at level $j < i$ and is charged the edge (u, v). This implies that level of u is at least j as well (since we charge the edge to a lower level). In particular, $u \in \tilde{G}_j$. Since $\deg_{\tilde{G}_{j-1}}(v) \le d_j$, this implies the total charge on v is at most d_j. Consider the set $\tilde{S}_j$, which is sampled from $\tilde{G}_j$. The total charge on $\tilde{S}_j$ is at most $\frac{100\cdot d_j\cdot|\tilde{V}_j|\cdot\log n}{d_{j+1}}$.

Finally, consider $v \in \tilde{V}_i$, if v is charged for an edge (u, v), then we conclude that $u \in \tilde{V}_i$ as well. So, the total charge on $\tilde{V}_i$ is at most the total number of edges in $\tilde{G}_i \cap \mathsf{Stream}_i$. Observe that $\deg_{\tilde{G}_{i-1}}(v) \le d_i$ for all $v \in \tilde{V}_i$. This implies that $\deg_{\tilde{G}_i}(v) \le d_i$ for all $v \in \tilde{V}_i$ (from Claim 11) with high probability. In the rest of the argument, we condition on this event. Let Y_v be a random variable denoting $|E_{\tilde{G}_i}(v) \cap \mathsf{Stream}_i|$. Note that Y_v is a sum of negatively associated 0-1 indicator random variables $\left\{X_e \mid e \in E_{\tilde{G}_i}(v)\right\}$, which takes value 1 if $e \in \mathsf{Stream}_i$ and 0 otherwise. We have the following bounds on probabilities for $e \in E_{\tilde{G}_i}(v)$

$$\Pr\left(X_e = 1\right) = \left(\frac{1}{m}\right)\left(\frac{100\cdot m\cdot\log n}{d_{i+1}}\right)$$

The above bound is valid since $d_r > 100\log n$ by our choice of parameters. Therefore, the bound on expectation of Y_v is as follows.

$$\mathbb{E}\left[Y_v\right] = \left(\frac{\deg_{\tilde{G}_i}(v)}{m}\right)\left(\frac{100\cdot m\cdot\log n}{d_{i+1}}\right) \le \left(\frac{d_i}{m}\right)\left(\frac{100\cdot m\cdot\log n}{d_{i+1}}\right) = \frac{100\cdot d_i\cdot\log n}{d_{i+1}}$$

We now apply Proposition 3 with $\mu_{\max} = \frac{100 \cdot d_i \cdot \log n}{d_{i+1}}$ and $\delta = {}^1\!/_2$ to get the bound:

$$\Pr\left(Y_v \geq \frac{150 \cdot d_i \cdot \log n}{d_{i+1}}\right) \leq \exp\left(-({}^5\!/_{10})^2 \left(\frac{100 \cdot d_i \cdot \log n}{d_{i+1}}\right)({}^1\!/_3)\right)$$
$$= O\left(\frac{1}{n^8}\right)$$
$$\text{(We use the fact that } d_i > d_{i+1})$$

Therefore, the total charge on $\tilde{V}_i$ is at most $\frac{150 \cdot d_i |\tilde{V}_i| \cdot \log n}{d_{i+1}}$. So from the above discussion it follows that the total number of edges stored in the memory till the end of Phase i is at most $\frac{100 \cdot n^2 \cdot \log n}{d_1} + \sum_{j=1}^{i-1} \frac{100 \cdot |\tilde{V}_j| \cdot d_j \cdot \log n}{d_{j+1}} + \frac{150 \cdot |\tilde{V}_i| \cdot d_i \cdot \log n}{d_{i+1}}$. From this we conclude that the total number of edges stored till the end of the stream $O(\frac{n^2 \cdot \log n}{d_1} + \sum_{i=1}^{r-1} \frac{n \cdot d_i \cdot \log n}{d_{i+1}} + n \cdot d_r)$, which proves our claim. ◀

▶ **Lemma 14.** *The total space complexity of Algorithm 3 is at most $\tilde{O}(n)$.*

Proof. From Lemma 13, the total number of edges stored in the memory at any time is $O(\frac{n^2 \cdot \log n}{d_1} + \sum_{i=1}^{r-1} \frac{n \cdot d_i \cdot \log n}{d_{i+1}} + n \cdot d_r)$. Since we chose $r = \log n - \log\log n - 7$ and $d_i = {}^n\!/_{2^i}$, we get the stated bound. ◀

(α, β)-Ruling sets in Random Order Streams. Algorithm 3 allows us to compute (α, β)-ruling sets for all $\beta \geq \alpha \geq 2$ in $\tilde{O}(n)$ space: in the final step, we just compute an$(\alpha, \alpha - 1)$ ruling set of $G[\cup_{j=0}^{r-1} \tilde{S}_j \cup \tilde{V}_r]$. We formally state the algorithm, and prove its correctness and space complexity in the full version of the paper.

4 Ruling Sets in Adversarial Streams

We also give an improved streaming algorithm that computes the β-ruling set of a graph in adversarial streams in $\tilde{O}(\beta \cdot n^{1+\frac{1}{2^\beta - 1}})$ space. This is significant for $\beta = 2$, since the best known algorithm for 2-ruling sets had space complexity $\tilde{O}(n^{3/2})$, while our analysis improves it to $\tilde{O}(n^{4/3})$. We state our main result for this section.

▶ **Theorem 15.** *There is an $\tilde{O}(\beta \cdot n^{1+\frac{1}{2^\beta - 1}})$-space streaming algorithm that computes a β-ruling set of any graph G with high probability. This implies an $\tilde{O}(n^{4/3})$-space streaming algorithm for computing a 2-ruling set of a graph G.*

4.1 A Slightly Improved Algorithm for Ruling Sets

Our algorithm does hierarchical sampling as described in the papers of [24, 25, 10]. However, we get a better bound on the space complexity. We illustrate the difference between their technique and ours for the simpler case of $\beta = 3$.

Comparison with Previous Work. Similar to [24, 25, 10], sample a set S_1 uniformly at random from V, and a set S_2 uniformly at random from S_1. During the stream, we collect edges incident on $G[S_1 \cup S_2]$. These sets are not modified during the stream in the earlier algorithms. On the other hand, we remove from S_1 all vertices that have a large degree in $G[S_1 \cup S_2]$, because we hope that for such vertices, one of its neighbours in $G[S_1 \cup S_2]$ will be included in S_2. This enables us to get a better bound on the degrees of vertices in $G[S_1 \cup S_2]$, and in turn a better bound on the space complexity. We note that [10, 25] state their peeling decomposition in a different way, however, their algorithm, too, peels off vertices that have higher "overall" degree (this is made explicit in Lemma 2.1 and Lemma 2.2 of [25]). Instead we remove vertices that have higher degree in the graph remaining after previous peeling steps. For completeness we now describe the sampling scheme.

▶ **Definition 16** (Sampling Scheme). *Given integers $s_1, s_2, \cdots, s_{\beta-1}$ such that $s_1 > s_2 > \cdots > s_{\beta-1} > 0$, we sample sets $S_1, \cdots, S_{\beta-1}$ as follows.*

1. *Let $S_0 = V$ and for $i \geq 1$, sample S_i uniformly at random from S_{i-1} and let $|S_i| = s_i$.*

For a vertex v we define the level of v, denoted $l(v)$ to be the unique index l such that $v \in S_l$, but $v \notin S_{l+1}$. If $v \in S_{\beta-1}$, then we say that level of v, denoted $l(v)$ is $\beta - 1$.

We now describe our algorithm.

Algorithm 4 Computing a β-Ruling Set in Adversarial Streams.

```
  Input: Take as input $(s_1, s_2, \cdots, s_{\beta-1})$, where $s_1 > s_2 > \cdots > s_{\beta-1}$
1 $S_0 \leftarrow V$ and $s_0 \leftarrow n$.
2 Sample $S_1, S_2, \cdots, S_{\beta-1}$ with parameters $s_1, \cdots, s_{\beta-1}$ according to Definition 16.
3 $\tilde{S}_i \leftarrow S_i$ for $i \in \{0, 1, \cdots, \beta-1\}$. We store all edges in $G[\cup_{j=0}^{\beta-1} \tilde{S}_j]$ in the stream.
4 For a vertex $v \in \cup_{j=0}^{\beta-1} \tilde{S}_j$, with $l(v) = l$ such that $l \leq \beta - 2$, if $\deg_{S_l}(v) \geq \frac{100 \cdot s_l \cdot \log n}{s_{l+1}}$,
    then let $\tilde{S}_j \leftarrow \tilde{S}_j \setminus \{v\}$ for all $j \leq l$. Delete any edges not in $G[\cup_{j=0}^{\beta-1} \tilde{S}_j]$.
5 Output MIS $\mathcal{M}$ of $G[\cup_{j=0}^{\beta-1} \tilde{S}_j]$.
```

We first claim that the algorithm indeed outputs a β-ruling set of G with high probability. Towards this, we prove the following claim.

▶ **Lemma 17.** *Consider any $v \in V$, then v is β-ruled by $\mathcal{M}$.*

Proof. We prove a stronger claim: We show that for all $i \in \{0, 1, \cdots, \beta-1\}$ each vertex $v \in S_i$ is $\beta - i$ ruled by $\mathcal{M}$. We prove this by induction. We first consider $S_{\beta-1}$ and start with the observation that $S_{\beta-1} \subseteq \cup_{j=1}^{\beta-1} \tilde{S}_j$ (this is because in Algorithm 4 Step 4 we only delete vertices of level less than $\beta - 1$ from $\cup_{j=1}^{\beta-1} \tilde{S}_j$). Since $\mathcal{M}$ is an MIS, this implies that for every $v \in S_{\beta-1}$, either v itself is in $\mathcal{M}$ or there is a neighbour u of v such that $u \in \mathcal{M}$. So, these vertices are 1-ruled.

We assume the claim holds for all $v \in S_i$ for some $i < \beta$. Under this inductive hypothesis, we want to prove this claim for all $v \in S_{i-1}$. Observe that if $l(v) \geq i$, then we know that $v \in S_i$ as well, and statement of the lemma holds by inductive hypothesis. So, we assume that $l(v) = i - 1$. Additionally, if $v \in \cup_{j=1}^{\beta-1} \tilde{S}_j$, then we know that v is 2-ruled. So, we assume that $v \notin \tilde{S}_{i-1}$. We therefore conclude that $\deg_{S_{i-1}}(v) \geq \frac{100 \cdot s_{i-1} \cdot \log n}{s_i}$. We now show that with high probability there is $u \in N_{S_{i-1}}(v)$ such that $u \in S_i$. Let Y_v be a random variable denoting $|N_{S_{i-1}}(v) \cap S_i|$. This random variable is the sum of negatively associated indicator random variables X_w for $w \in N_{S_{i-1}}(v)$, which takes value 1 if $w \in S_i$ and 0 otherwise. We have the following probability bound.

$$\Pr(X_w = 1) = \left(\frac{s_i}{s_{i-1}}\right)$$

$$\mathbb{E}[Y_v] = \deg_{S_{i-1}}(v) \cdot \left(\frac{s_i}{s_{i-1}}\right) \geq 100 \log n$$

Applying Proposition 3 with $\mu_{\min} = 100 \log n$ and $\delta = \frac{1}{2}$, we have:

$$\Pr\left(X_v \leq 50 \log n\right) = \exp\left(-({}^1\!/_2)^2 (100 \log n)({}^1\!/_2)\right) = O\left(\frac{1}{n^{12}}\right)$$

$$\Pr\left(X_v > 50 \log n\right) = 1 - O\left(\frac{1}{n^{12}}\right).$$

This implies that with high probability, v has a neighbour in S_i. By inductive hypothesis, all vertices in S_i are at most $\beta - i$ ruled by $\mathcal{M}$, this implies that v is $\beta - i + 1$ ruled by $\mathcal{M}$. ◀

We now bound the total number of edges stored in the memory at any time.

▶ **Lemma 18.** *The total number of edges stored in the memory at any point in time is at most $\tilde{O}({}^{n^2}\!/_{s_1} + \sum_{j=2}^{\beta-1} {}^{s_{j-1}^2}\!/_{s_j} + s_{\beta-1}^2)$.*

Proof. To prove this claim, we do a charging argument. We charge every edge (u, v) to the vertex which is at a lower level, breaking ties arbitrarily. Consider a vertex $v \in \cup_{j=0}^{\beta-1} \tilde{S}_j$ and suppose $l(v) = i$. This implies that $\deg_{S_i}(v) \leq \frac{100 \cdot s_i \cdot \log n}{s_{i+1}}$. So, the total charge on vertices $v \in \cup_{j=0}^{\beta-1} \tilde{S}_j$ with level i is at most $\tilde{O}({}^{s_i^2}\!/_{s_{i+1}})$. If a vertex $v \in S_{\beta-1}$ is charged for an edge (u, v), then $u \in S_{\beta-1}$ as well. So, the total bound on the number of edges is $\tilde{O}({}^{n^2}\!/_{s_1} + \sum_{j=2}^{\beta-1} {}^{s_{j-1}^2}\!/_{s_j} + s_{\beta-1}^2)$. ◀

▶ **Lemma 19.** *The space complexity of Algorithm 4 is at most $\tilde{O}(\beta \cdot n^{1+{}^1\!/_{2^\beta-1}})$.*

Proof. Consider the bound on the edges in Lemma 18, we let $s_i = n^{1-\frac{2^i-1}{2^\beta-1}}$ for $i \in \{0, 1, \cdots, \beta - 1\}$. Consider the term $\frac{s_i^2}{s_{i+1}}$ in the sum, we have the following bound on it.

$$\frac{s_i^2}{s_{i+1}} = \frac{n^{2-\frac{2^{i+1}-2}{2^\beta-1}}}{n^{1-\frac{2^{i+1}-1}{2^\beta-1}}} = n^{1+\frac{1}{2^\beta-1}}.$$

Similarly, $s_{\beta-1}^2 = n^{2-\frac{2^\beta-2}{2^\beta-1}} = n^{1+\frac{1}{2^\beta-1}}$.This proves our claim. ◀

Our analysis gives an improved bound of $\tilde{O}(n^{4/3})$ for 2-ruling set. The previous best known algorithm had a space bound of $\tilde{O}(n^{3/2})$.

(α, β)-Ruling sets in a Adversarial Order Streams. A modification of Algorithm 4 allows us to compute (α, β)-ruling sets for all $\beta \geq \alpha \geq 2$ in $\tilde{O}(n^{1+\frac{1}{2^{\beta-\alpha+2}-1}})$ space: in the final step, we just compute an $(\alpha, \alpha - 1)$ ruling set instead of MIS. We give the algorithm, along with a proof of its space bound and correctness in the full version of the paper

4.2 Lower Bound for Ruling Sets

In this section, our goal is to prove the following theorem.

▶ **Theorem 20.** *Every randomized constant error one-pass streaming algorithm in the adversarial edge arrival model that computes an $(\alpha, \alpha - 1)$-ruling set in an n-vertex graph for any even $\alpha > 2$, requires $\Omega({}^{n^2}\!/_{\alpha^2})$ space.*

We now give a hard distribution $\mathcal{D}_{\mathsf{Rule}}$ for our algorithm (refer to Figure 1).

Distribution 5 **Distribution $\mathcal{D}_{\mathsf{Rule}}$.**

```
Output: An instance $(G_{1/2}, a, b)$.
1 Let $\alpha := 2c$ for $c \geq 2$ be an even number and let $n = 2c \cdot N + 4c \cdot (c-1)$. Let $G_{1/2}$ be a random graph on $N$ vertices with each edge being included with probability $\frac{1}{2}$
2 Create $\alpha$ disjoint copies $G_1, \cdots, G_\alpha$ of $G_{1/2}$.
3 Create $\alpha$ paths $P_x^i$ with vertices $x_1^i, \cdots, x_{c-1}^i$ and $P_y^i$ with vertices $y_1^i, \cdots, y_{c-1}^i$ for $i \in [\alpha]$.
4 Pick a pair of vertices $(a, b)$ of $G_{1/2}$ uniformly at random. Let $(a_i, b_i)$ be the copy of $(a, b)$ in $G_i$ for $i \in [\alpha]$.
5 For all $i \in [\alpha]$, add an edge between $a_i$ and $x_1^i$, and between $b_i$ and $y_1^i$.
6 For all $i \in [\alpha - 1]$, add an edge between each $w \in V(G_i) \setminus \{a_i, b_i\}$ and $z \in V(G_{i+1}) \setminus \{a_{i+1}, b_{i+1}\}$. Similarly, add an edge between each $w \in V(G_1) \setminus \{a_1, b_1\}$ and $z \in V(G_\alpha) \setminus \{a_\alpha, b_\alpha\}$.
```

To relate the ruling set problem with Index_t, we will condition on the following event.

- **Event** $\mathcal{E}_{\text{DIST}}$. For all $u, v \in G_{1/2}$, $\text{dist}(u, v) \leq 2$.

We state the following claim.

▷ Claim 21. $\Pr(\mathcal{E}_{\text{DIST}}) \geq 1 - o(1)$.

Proof sketch. To bound the probability of $\mathcal{E}_{\text{DIST}}$, it is sufficient to bound the probability that there is a pair of vertices which don't share a common neighbour. Consider two vertices u and v, the probability that a fixed vertex $w \notin N(u) \cap N(v)$ is $3/4$. Therefore, the probability that $N(u) \cap N(v) = \emptyset$ is at most $(3/4)^N$. Thus, taking a union bound over all pairs of vertices, we have the desired claim. ◁

From now on, while discussing the properties of G, we condition on the event $\mathcal{E}_{\text{DIST}}$.

Before moving on to the proof of Theorem 20, we first clarify some notation. In what follows, $G := (G_{1/2}, a, b)$ will denote a graph sampled from $\mathcal{D}_{\mathsf{Rule}}$, and $V(G)$ will denote the vertices of G (This includes vertices of G_i, P_x^i and P_y^i for all $i \in [\alpha]$). The graph G_i for $i \in [\alpha]$ will denote the ith copy of $G_{1/2}$ in G, and $V(G_i)$ denotes its vertex set. The set $V(G_i)$ does not include vertices of P_x^i and P_y^i. We use $V(G_i \cup P_x^i \cup P_y^i)$ to denote the vertices of G_i, P_x^i, and P_y^i for all $i \in [\alpha]$.

Towards proving our main theorem, we state a few properties of $(G_{1/2}, a, b)$. It might help to refer to Figure 1 for the following claim.

▷ Claim 22. Let $G := (G_{1/2}, a, b)$ be drawn from $\mathcal{D}_{\mathsf{Rule}}$. For any $j \in [\alpha]$, consider any $u \in V(G_j) \setminus \{a_j, b_j\}$ and $v \in V(G) \setminus \{x_k^{c-1}, y_k^{c-1}\}$ for $k \equiv c + j \mod \alpha$; then $\text{dist}(u, v) \leq 2c - 1 = \alpha - 1$. Moreover $\text{dist}(u, x_k^{c-1}) = 2c = \alpha$ and $\text{dist}(u, y_k^{c-1}) = 2c = \alpha$.

Proof. We prove the claim for $j = 1$, and the proof is identical for all other values of j. We consider $u \in V(G_1) \setminus \{a_1, b_1\}$; we want to argue that u is at distance at most $2c - 1$ from all $v \in V(G) \setminus \{x_{c+1}^{c-1}, y_{c+1}^{c-1}\}$. We consider the following cases.

1. Consider any $v \in V(G_i) \setminus \{a_i, b_i\}$ for $i \leq c + 1$, then $\text{dist}(u, v) \leq c$. If $v \in V(G_1)$ then $\text{dist}(u, v) \leq 2$ since we condition on $\mathcal{E}_{\text{DIST}}$. If $v \in V(G_i) \setminus \{a_i, b_i\}$ for $i \neq 1$, u and v are connected by a path of length $i - 1$. For $i \geq c + 1$, the argument is identical

2. We can reach the vertices x_c^{c-1} and y_c^{c-1} in $2c-1$ hops from u: since we condition on $\mathcal{E}_{\text{DIST}}$, there is a $w \in V(G_c) \setminus \{a_c, b_c\}$ such that $\text{dist}(x_c^{c-1}, w) = c$. Moreover, $d(u, w) = c - 1$. This implies that all vertices of $V(G_i \cup P_x^i \cup P_y^i)$ for $i \leq c$ are $2c - 1$ ruled by u. The proof for all $i \geq c + 2$ is identical.
3. Consider the vertices x_{c+1}^{c-1} and y_{c+1}^{c-1}. For these vertices, $\text{dist}(u, x_{c-1}^{c+1}) = 2c$ and $\text{dist}(u, y_{c-1}^{c+1}) = 2c$. Additionally, for all other vertices in the set $v \in V(G_{c+1} \cup P_x^{c+1} \cup P_y^{c+1}) \setminus \left\{x_{c+1}^{c-1}, y_{c+1}^{c-1}\right\}$, $\text{dist}(u, v) \leq 2c - 1$.

◁

▶ **Lemma 23.** *Let I be a $(\alpha, \alpha - 1)$-ruling set of $(G_{1/2}, a, b)$. Suppose for some $i \in [2c]$, $x_i^{c-1} \in I$; if $(a, b) \notin E(G_{1/2})$, then $y_i^{c-1} \in I$ as well.*

Proof. We show the claim for $i = 1$, and for other values, the proof is identical. To prove this claim, we show that the vertices that $(2c - 1)$-rule x_1^{c-1} and y_1^{c-1} are the same. Let $A_1 = \left\{v \notin \left\{x_1^{c-1}, y_1^{c-1}\right\} \mid \text{dist}(v, x_1^{c-1}) = 2c - 1\right\}$. These are the vertices that $2c-1$ rule x_1^{c-1}, not including y_1^{c-1}. Similarly we define $A_2 = \left\{v \notin \left\{x_1^{c-1}, y_1^{c-1}\right\} \mid \text{dist}(v, y_1^{c-1}) = 2c - 1\right\}$. We claim that $A_1 = A_2$. We consider the following cases.

1. For any x_1^k for $k \leq c - 2$, $\text{dist}(x_1^k, y_1^{c-1}) \leq 2c - 1$ and similarly, $\text{dist}(x_1^k, y_1^{c-1}) \leq 2c - 1$. This shows that $x_1^k \in A_1$ and $x_1^k \in A_2$ as well. Similarly, we can show that for $k \leq c - 2$, $y_1^k \in A_1$ and $y_1^k \in A_2$.
2. For any vertex $v \in V(G_1)$, $\text{dist}(v, x_1^{c-1}) \leq c + 1$ and similarly, $\text{dist}(v, y_1^{c-1}) \leq c + 1$ (since we condition on $\mathcal{E}_{\text{DIST}}$), and therefore, $v \in A_1$ and $v \in A_2$.
3. We now want to consider all vertices $v \notin V(G_1 \cup P_x^1 \cup P_y^1)$. Since we condition on $\mathcal{E}_{\text{DIST}}$, we know that there exists a vertex $u \in V(G_1) \setminus \{a_1, b_1\}$, such that $\text{dist}(u, x_1^{c-1}) = \text{dist}(u, y_1^{c-1}) = c$. Therefore, for any $v \notin V(G_1 \cup P_x^1 \cup P_y^1)$, $\text{dist}(v, x_1^{c-1}) = \text{dist}(v, y_1^{c-1})$.

From the above three cases, we conclude that $A_1 = A_2$. So, if $x_1^{c-1} \in I$, then for all $w \in A_2$, $w \notin I$. Since $(a, b) \notin E(G_{1/2})$, this implies that $\text{dist}(x_1^{c-1}, y_1^{c-1}) = 2c$. So, $y_1^{c-1} \in I$. ◀

▷ Claim 24. For all $i \in [\alpha]$, we denote a_i and b_i by x_i^0 and y_i^0. Let I be a $(\alpha, \alpha - 1)$-ruling set of $(G_{1/2}, a, b)$. Suppose $x_i^k \in I$ for some $k \leq c - 2$. If $(a, b) \notin E(G_{1/2})$, then $\left\{x_j^{c-1}, y_j^{c-1}\right\} \subseteq I$ for $j \equiv (i + c - k - 1) \mod \alpha$.

Proof. Consider x_j^{c-1}; there is a vertex $w \in V(G_j) \setminus \{a_j, b_j\}$ such that $\text{dist}(w, x_j^{c-1}) = c$ and there is a vertex $z \in V(G_i) \setminus \{a_i, b_i\}$ such that $\text{dist}(z, x_i^k) = k + 1$. Additionally, by our construction, $\text{dist}(w, z) = j - i$. This implies that: $\text{dist}(x_i^k, x_j^{c-1}) = c + (k + 1) + (j - i) = 2c$. We conclude that both vertices x_j^{c-1} and y_j^{c-1} are not $(2c - 1)$-ruled by x_i^k.

Additionally, for all $k < j$, all vertices in $V(G_k \cup P_x^k \cup P_y^k)$ are excluded from being added in I, since all these vertices are $(2c - 1)$-ruled by x_i^k. Similarly, all vertices in $V(G_j \cup P_x^j \cup P_y^j) \setminus \left\{x_j^{c-1}, y_j^{c-1}\right\}$ are $(2c - 1)$-ruled by x_i^k, so these vertices are excluded from I as well. Finally, we argue that all vertices in $V(G_k \cup P_x^k \cup P_y^k)$ for $k \geq j + 1$ that could $(2c - 1)$-rule x_j^{c-1} and y_j^{c-1} are excluded from I. We prove this claim for $k = j + 1$, but it is identical for all other values of k.

In $V(G_{j+1} \cup P_x^{j+1} \cup P_y^{j+1})$, the vertices that can be included in I are $\left\{x_{j+1}^{c-2}, y_{j+1}^{c-2}, x_{j+1}^{c-1}, y_{j+1}^{c-1}\right\}$, however $\text{dist}(x_{j+1}^{c-2}, x_j^{c-1}) = (c - 1) + 1 + c = 2c$. Therefore, we conclude that all the vertices other than x_j^{c-1} and y_j^{c-1} that are still candidates for I cannot $(2c - 1)$-rule x_j^{c-1} and y_j^{c-1}. Additionally, since $\text{dist}(x_j^{c-1}, y_j^{c-1}) = 2c$, this implies that $\left\{x_j^{c-1}, y_j^{c-1}\right\} \subseteq I$. ◁

▶ **Lemma 25.** *Let I be any $(\alpha, \alpha - 1)$-ruling set of $(G_{1/2}, a, b)$. Then, there is an $i \in [\alpha]$ such that $\left\{x_i^{c-1}, y_i^{c-1}\right\} \subseteq I$ if and only if $(a, b) \notin G$.*

Proof. Suppose $(a,b) \in G_{1/2}$, then for any $i \in [\alpha]$, $\text{dist}(x_i^{c-1}, y_i^{c-1}) = 2c-1$, so, both cannot be included in I. Consider the other direction when $(a,b) \notin G_{1/2}$. Let I be any $(\alpha,\alpha-1)$-ruling set of $(G_{1/2}, a, b)$ and let $v \in I$. We consider the following cases for v.

1. **$v \in V(G_i) \setminus \{a_i, b_i\}$ for some $i \in [\alpha]$.** In this case, from Claim 22, we conclude that x_k^{c-1} and y_k^{c-1} for $k \equiv i + c \mod \alpha$ are the only two vertices that are not ruled. Additionally, since $\text{dist}(x_k^{c-1}, y_k^{c-1}) = \alpha$, we conclude that $\{x_k^{c-1}, y_k^{c-1}\} \subseteq I$.
2. **$v = x_i^{c-1}$ or $v = y_i^{c-1}$ for some $i \in [\alpha]$.** In this case, Lemma 23, implies that $\{x_i^{c-1}, y_i^{c-1}\} \subseteq I$.
3. **$v = x_i^k$ or $v = y_i^k$ for some $k \leq c-2$.** In this case, from Claim 24, we conclude that $\{x_j^{c-1}, y_j^{c-1}\} \subseteq I$ for $j \equiv (i + c - k - 1) \mod 2c$.

This proves our lemma. ◀

We now state the next lemma, which will imply Theorem 20. Throughout this section, $\mathcal{A}_{\mathsf{Rule}}$ is an algorithm that for an even $\alpha > 2$ computes an $(\alpha, \alpha-1)$ ruling set with error at most δ when the input is sampled from $\mathcal{D}_{\mathsf{Rule}}$.

▶ **Lemma 26.** *There exists a $(\delta + o(1))$-error protocol π_{Ind} for Index_t on $\mathcal{D}_{\mathsf{Ind}}$ such that the communication cost of π_{Ind} is $s(\mathcal{A}_{\mathsf{Rule}})$, where $s(\mathcal{A}_{\mathsf{Rule}})$ is the space complexity of $\mathcal{A}_{\mathsf{Rule}}$.*

Proof. We begin the proof by designing the protocol π_{Ind} (see Protocol 6). From now on, $t = \binom{N}{2}$ and $n = 2c \cdot N + 4c(c-1)$.

Protocol 6 Protocol π_{Ind}.

```
Input: An instance $(X, \sigma) \sim \mathcal{D}_{\mathsf{Ind}}$
Output: Yes if $X_\sigma = 1$ and No if $X_\sigma = 0$.
1 Alice creates an $N$-vertex graph $H$ whose adjacency matrix is given by $X$.
2 She then creates $\alpha$ disjoint copies of $H$: $G_1, G_2, \cdots, G_\alpha$.
3 Alice then creates $4\alpha$ paths on $c-1$ vertices: $\{P_x^i\}_{1 \leq i \leq \alpha}$ and $\{P_y^i\}_{1 \leq i \leq \alpha}$.
4 Bob treats $\sigma \in [t]$ as an edge $(a,b)$ of an $n$-vertex graph. He adds edges $(a_i, x_i^1)$, and
   $(b_i, y_i^1)$ for all $i \in [\alpha]$. Additionally, for all $i \in [\alpha - 1]$, he adds edges $(x,y)$ for all
   $x \in V(G_i) \setminus \{a_i, b_i\}$ and for all $y \in V(G_{i+1}) \setminus \{a_{i+1}, b_{i+1}\}$. Edges $(x,y)$ are also
   added for all $x \in V(G_1) \setminus \{a_1, b_1\}$ and $y \in V(G_{2c}) \setminus \{a_{2c}, b_{2c}\}$.
5 The players compute an $(\alpha, \alpha-1)$-ruling set on their graph using $\mathcal{A}_{\mathsf{Rule}}$ and output
   No if there is an $i \in [\alpha]$ such that $\{x_i^{c-1}, y_i^{c-1}\} \subseteq I$, where $I$ is an $(\alpha, \alpha-1)$-ruling
   set. Output Yes if there is no such $i \in [\alpha]$.
```

The distribution of (G, a, b) created by π_{Ind} matches the distribution $\mathcal{D}_{\mathsf{Rule}}$ exactly. From Lemma 25 and using the fact that $X_\sigma = 1$ if and only if $(a,b) \in E(G)$, we conclude that π_{Ind} outputs Yes if $X_\sigma = 1$ and No if $X_\sigma = 0$. Since the event $\neg\mathcal{E}_{\text{DIST}}$ happens with probability $o(1)$, we conclude that: $\Pr(\pi_{\mathsf{Ind}} \text{ errs}) \leq \Pr(\mathcal{A}_{\mathsf{Rule}} \text{ errs}) + \Pr(\neg\mathcal{E}_{\text{DIST}}) = \delta + o(1)$. Since Alice sends the memory contents of $\mathcal{A}_{\mathsf{Rule}}$, we know that the communication cost of π_{Ind} is $s(\mathcal{A}_{\mathsf{Rule}})$. ◀

Proof of Theorem 20. If there was an $o(n^2/\alpha^2)$-space δ-error one-pass streaming algorithm for the problem of finding a $(\alpha, \alpha-1)$-ruling set of a graph G, then Alice and Bob would be able to solve Index_t using this algorithm. Alice would run it on her input, and send the contents of the memory to Bob, who would run it on his input and give the ruling set output by the algorithm. Since the contents of the memory are $o(n^2/\alpha^2)$, this would give a protocol for Index_t with communication complexity $o(n^2/\alpha^2) = o(t)$ and $(\delta + o(1))$-error, thus contradicting Proposition 5. ◀

References

1 Amir Abboud, Keren Censor-Hillel, Seri Khoury, and Ami Paz. Smaller cuts, higher lower bounds. *arXiv preprint*, 2019. arXiv:1901.01630.

2 Kook Jin Ahn, Graham Cormode, Sudipto Guha, Andrew McGregor, and Anthony Wirth. Correlation clustering in data streams. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 2237–2246. JMLR.org, 2015.

3 Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. *Analyzing Graph Structure via Linear Measurements*, pages 459–467. SIAM, 2012. doi:10.1137/1.9781611973099.40.

4 Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1616–1635. SIAM, 2019. doi:10.1137/1.9781611975482.98.

5 Sepehr Assadi and Soheil Behnezhad. Beating two-thirds for random-order streaming matching. *CoRR*, abs/2102.07011, 2021. arXiv:2102.07011.

6 Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for $(\Delta + 1)$ vertex coloring. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 767–786. SIAM, 2019. doi:10.1137/1.9781611975482.48.

7 B. Awerbuch, M. Luby, A.V. Goldberg, and S.A. Plotkin. Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science*, pages 364–369, 1989. doi:10.1109/SFCS.1989.63504.

8 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3), 2016. doi:10.1145/2903137.

9 Aaron Bernstein. Improved bounds for matching in random-order streams. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 12:1–12:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ICALP.2020.12.

10 Tushar Bisht, Kishore Kothapalli, and Sriram Pemmaraju. Brief announcement: Super-fast t-ruling sets. In *PODC*, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2611462.2611512.

11 Amit Chakrabarti, Graham Cormode, and Andrew McGregor. Robust lower bounds for communication and stream computation. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, page 641–650, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1374376.1374470.

12 Yi-Jun Chang, Wenzheng Li, and Seth Pettie. An optimal distributed $(\delta + 1)$-coloring algorithm? In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, page 445–456, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3188745.3188964.

13 Graham Cormode, Jacques Dark, and Christian Konrad. Independent sets in vertex-arrival streams. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 45:1–45:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.45.

14 Devdatt P. Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998. doi:10.1002/(SICI)1098-2418(199809)13:2\%3C99::AID-RSA1\%3E3.0.CO;2-M.

15 Alireza Farhadi, MohammadTaghi Hajiaghayi, Tung Mai, Anup Rao, and Ryan A. Rossi. Approximate maximum matching in random streams. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, page 1773–1785, USA, 2020. Society for Industrial and Applied Mathematics.

16 Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 491–500, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331603.

17 Beat Gfeller and Elias Vicari. A randomized distributed algorithm for the maximal independent set problem in growth-bounded graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 53–60, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281100.1281111.

18 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, page 270–277, USA, 2016. Society for Industrial and Applied Mathematics.

19 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '16, page 489–498, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2897518.2897638.

20 H. Jowhari, M. Saglam, and G. Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. *arXiv*, abs/1012.4889, 2011. arXiv:1012.4889.

21 Tomasz Jurdziński and Krzysztof Nowicki. Mst in o(1) rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, page 2620–2632, USA, 2018. Society for Industrial and Applied Mathematics.

22 Christian Konrad. MIS in the congested clique model in o(log log Δ) rounds. *CoRR*, abs/1802.07647, 2018. arXiv:1802.07647.

23 Christian Konrad, Frédéric Magniez, and Claire Mathieu. Maximum matching in semi-streaming with few passes. In Anupam Gupta, Klaus Jansen, José D. P. Rolim, and Rocco A. Servedio, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 15th International Workshop, APPROX 2012, and 16th International Workshop, RANDOM 2012, Cambridge, MA, USA, August 15-17, 2012. Proceedings*, volume 7408 of *Lecture Notes in Computer Science*, pages 231–242. Springer, 2012. doi:10.1007/978-3-642-32512-0_20.

24 Christian Konrad, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson. The Complexity of Symmetry Breaking in Massive Graphs. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2019.26.

25 Kishore Kothapalli and Sriram V. Pemmaraju. Super-fast 3-ruling sets. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, volume 18 of *LIPIcs*, pages 136–147. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPIcs.FSTTCS.2012.136.

26 Ilan Kremer, Noam Nisan, and Dana Ron. On randomized one-round communication complexity. In Frank Thomson Leighton and Allan Borodin, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 596–605. ACM, 1995. doi:10.1145/225058.225277.

27 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *J. ACM*, 63(2):17:1–17:44, 2016. doi:10.1145/2742012.

28 Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Rec.*, 43(1):9–20, 2014. doi:10.1145/2627692.2627694.

29 Morteza Monemizadeh, S. Muthukrishnan, Pan Peng, and Christian Sohler. Testable bounded degree graph properties are random order streamable. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 131:1–131:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.131.

30 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 429–438, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2935764.2935785.

31 Johannes Schneider and Roger Wattenhofer. A new technique for distributed symmetry breaking. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, page 257–266, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1835698.1835760.

32 David Wajc. Negative association-definition, properties, and applications, 2017. URL: http://www.cs.cmu.edu/~dwajc/notes/Negative%20Association.pdf.

A Missing Figures

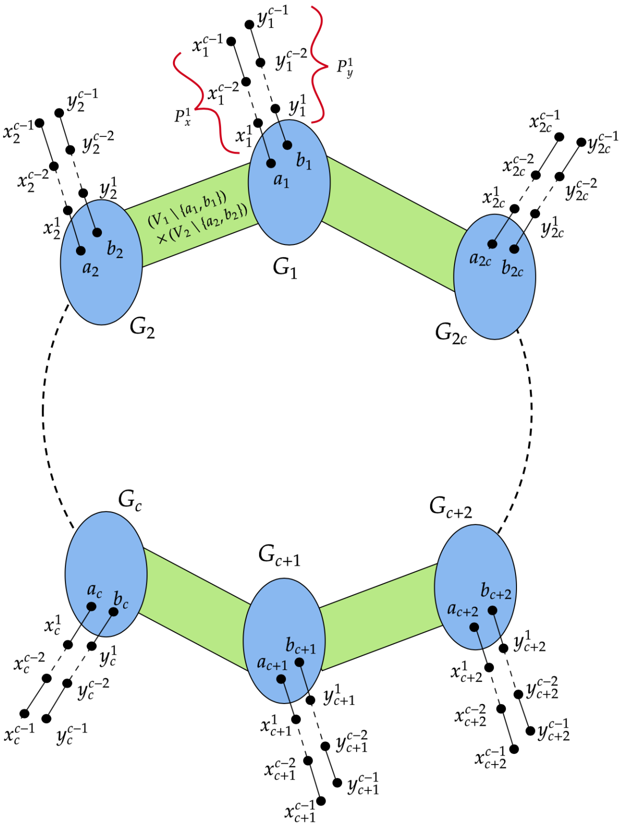


Figure 1 To create an instance $(G_{1/2}, a, b)$, graph $G_{1/2}$ is sampled by including every edge on N-vertices with probability $1/2$. Then $\alpha := 2c$ disjoint copies of $G_{1/2}$ are created. Additionally, $4c$ paths on $c-1$ vertices, $\left\{P_x^i\right\}_{1\leq i\leq 2c}$ and $\left\{P_y^i\right\}_{1\leq i\leq 2c}$ are created. A random edge (a, b) is sampled and $4c$ edges, $\left\{(a_i, x_i^1)\right\}_{1\leq i\leq 2c}$ and $\left\{(b_i, y_i^1)\right\}_{1\leq i\leq 2c}$ are added. Edges (x, y) for all $x \in V(G_i) \setminus \{a_i, b_i\}$ and $y \in V(G_{i+1}) \setminus \{a_{i+1}, b_{i+1}\}$ for all $i \in [2c-1]$ are added. Finally, edges (w, z) for all $w \in V(G_1) \setminus \{a_1, b_1\}$ and $z \in V(G_{2c}) \setminus \{a_{2c}, b_{2c}\}$ are also added.

Impossibility of Strongly-Linearizable Message-Passing Objects via Simulation by Single-Writer Registers

Hagit Attiya
Computer Science Department, Technion, Haifa, Israel

Constantin Enea
Université de Paris, IRIF, CNRS, France

Jennifer L. Welch
Texas A&M University, College Station, TX, USA

Abstract

A key way to construct complex distributed systems is through modular composition of linearizable concurrent objects. A prominent example is shared registers, which have crash-tolerant implementations on top of message-passing systems, allowing the advantages of shared memory to carry over to message-passing. Yet linearizable registers do not always behave properly when used inside randomized programs. A strengthening of linearizability, called strong linearizability, has been shown to preserve probabilistic behavior, as well as other "hypersafety" properties. In order to exploit composition and abstraction in message-passing systems, it is crucial to know whether there exist strongly-linearizable implementations of registers in message-passing. This paper answers the question in the negative: *there are no strongly-linearizable fault-tolerant message-passing implementations of multi-writer registers, max-registers, snapshots or counters.* This result is proved by reduction from the corresponding result by Helmi et al. The reduction is a novel extension of the BG simulation that connects shared-memory and message-passing, supports long-lived objects, and preserves strong linearizability. The main technical challenge arises from the discrepancy between the potentially minuscule fraction of failures to be tolerated in the simulated message-passing algorithm and the large fraction of failures that can afflict the simulating shared-memory system. The reduction is general and can be viewed as the inverse of the ABD simulation of shared memory in message-passing.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Concurrent Objects, Message-passing systems, Strong linearizability, Impossibility proofs, BG simulation, Shared registers

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.7

Related Version *Full Version*: `https://arxiv.org/abs/2105.06614`

Funding *Hagit Attiya*: Partially supported by the Israel Science Foundation (grant number 380/18).
Constantin Enea: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).
Jennifer L. Welch: Supported in part by the U.S. National Science Foundation under grant number 1816922.

Acknowledgements We thank the anonymous referees for helpful comments that improved the presentation of the paper.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 7; pp. 7:1–7:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A key way to construct complex distributed systems is through modular composition of linearizable concurrent objects [19]. A prominent example is the ABD fault-tolerant message-passing implementation of shared registers [3] and its multi-writer variant [24]. In multi-writer ABD, there is a set of client processes, which accept invocations of methods on the shared register and provide responses, and a set of server processes, which replicate the (virtual) state of the register. When a read method is invoked at a client, the client queries a majority of the servers to obtain the latest value, as determined by a timestamp, and then sends the chosen value back to a majority of the servers before returning. When a write method is invoked, the client queries a majority of the servers to obtain the latest timestamp, assigns a larger timestamp to the current value to be written, and then sends the new value and its timestamp to a majority of the servers before returning. Client and server processes run on a set of (physical) nodes; a node may run any combination of client and server processes. The algorithm tolerates any distribution of process crashes as long as less than half of the server processes crash; there is no limit on the number of client processes that crash.

Variations of ABD have been used to simplify the design of numerous fault-tolerant message-passing algorithms, by providing the familiar shared-memory abstraction (e.g., in *Disk Paxos* [13]). Yet linearizable registers do not always compose correctly with randomized programs: In particular, [17] demonstrates a randomized program that terminates with constant probability when used with an *atomic* register, on which methods execute instantaneously, but an adversary can in principle prohibit it from terminating when the register is implemented in a message-passing system, where methods are not instantaneous. The analogous result is shown in [6] specifically for the situation when ABD is the implementation.

Strong linearizability [14], a restriction of linearizability, ensures that properties holding when a concurrent program is executed in conjunction with an atomic object, continue to hold when the program is executed with a strongly-linearizable implementation of the object. Strong linearizability was shown [5] to be necessary and sufficient for preserving *hypersafety properties* [8], such as security properties and probability distributions of reaching particular program states.

These observations highlight the importance of knowing whether there exists a *strongly-linearizable* fault-tolerant message-passing implementation of a shared register. If none exists, it will be necessary to argue about hypersafety properties without being able to capitalize on the shared-memory abstraction.

This paper brings bad news, answering this question in the negative: *There are no strongly-linearizable fault-tolerant message-passing implementations of several highly useful objects, including multi-writer registers, max-registers, snapshots and counters.*

One might be tempted to simply conclude this result from the impossibility result of Helmi et al. [18] showing that there is no strongly-linearizable nonblocking implementation of a multi-writer register from single-writer registers. However, reproducing the proof in [18] for the message-passing model is not simple, as it is rather complicated and tailored to the shared-memory model. In particular, parts of the proof require progress when a process executes solo, which cannot be easily imitated when the number of failures is much smaller than the number of processes.

Another approach is to reduce to the impossibility result in the shared-memory model. A simple reduction simulates message transfer between each pair of message-passing nodes using dedicated shared registers. This simulation uses the same number of shared-memory processes as message-passing nodes and preserves the number of failures tolerated. However, message-passing register implementations require the total number of nodes to be at least twice the number of failures tolerated [3], while the proof of Helmi et al. critically depends on the fact that all processes, except perhaps one, may stop taking steps. It is not obvious how to simulate the workings of many message-passing nodes, only a small fraction of which may fail, using the same number of shared-memory processes, almost all of which may fail.

We take a different path to prove this result by reduction, extending the BG simulation [7] in three nontrivial ways. First, our reduction works across communication models, and bridges the gap between shared-memory and message-passing systems. Second, it supports long-lived objects, on which each process can invoke any number of methods instead of just one. And most importantly, it preserves strong linearizability.

In more detail, we consider a hypothetical strongly-linearizable message-passing algorithm that implements a long-lived object. Following contemporary expositions of such algorithms (e.g. [15, 20]), we assume that the algorithm is organized into a set of m client processes, any number of which may crash, and n server processes, up to $m-1$ of which may crash, running on a set of nodes. We obtain a nonblocking shared-memory implementation of the same object for m processes, $m-1$ of which may fail, using *single-writer* registers.

Our implementation admits a *forward simulation* to the message-passing implementation. The forward simulation is a relation between states of the two implementations. Using the forward simulation, we can construct an execution of the message-passing implementation from any execution of the shared-memory implementation, starting from the initial state and moving forward step by step, such that the two executions have the same sequence of method invocations and responses.

Since the hypothetical message-passing algorithm is strongly linearizable, a result from [5, 27] implies that there is a forward simulation from the message-passing algorithm to the atomic object. Since forward simulations compose, we obtain a forward simulation from the shared memory algorithm to the atomic object. Another result from [5, 27] shows that a forward simulation implies strong linearizability. Therefore, a strongly-linearizable message-passing implementation of a multi-writer register yields a strongly-linearizable shared-memory implementation of a multi-writer register using single-writer registers. Now we can appeal to the impossibility result of [18] to conclude that there can be no strongly-linearizable message-passing implementation of a multi-writer register. The same argument shows the impossibility of strongly-linearizable message-passing implementations of max-registers, snapshots, and counters, which are proved in [18] to have no strongly-linearizable implementations using single-writer registers.

We consider the reduction to be interesting in its own right, because it shows how general message-passing object implementations can be translated into corresponding shared-memory object implementations. In this sense, it can be interpreted as an inverse of ABD, which translates shared-memory object implementations into message-passing object implementations. It thus relates the two models, keeping the same number of failures, without restricting the total number of processes in the message-passing model. We believe it may have additional applications in other contexts.

2 Objects

An *object* is defined by a set of method names and an implementation that defines the behavior of each method. Methods can be invoked in parallel at different processes. The executions of an implementation are modeled as sequences of labeled transitions between global states that track the local states of all the participating processes (more precise definitions will be given in Section 2.2 and Section 2.3). Certain transitions of an execution correspond to new invocations of a method or returning from an invocation performed in the past. Such transitions are labeled by call and return actions, respectively. A *call action* $call\ M(x)_k$ represents the event of invoking a method M with argument x; k is an identifier of this invocation. A *return action* $ret\ y_k$ represents the event of the invocation k returning value y. For simplicity, we assume that each method takes as parameter or returns a single value. We may omit invocation identifiers from call or return actions when they are not important. The set of executions of an object O is denoted by $E(O)$.

2.1 Object Specifications

The specification of an object characterizes sequences of call and return actions, called *histories*. The history of an execution e, denoted by $hist(e)$, is defined as the projection of e on the call and return actions labeling its transitions. The set of histories of all the executions of an object O is denoted by $H(O)$. Call and return actions $call\ M(x)_k$ and $ret\ y_k$ are called *matching* when they contain the same invocation identifier k. A call action is called *unmatched* in a history h when h does not contain the matching return. A history h is called *sequential* if every call $call\ M(x)_k$ is immediately followed by the matching return $ret\ y_k$. Otherwise, it is called *concurrent*.

Linearizability [19] expresses the conformance of object histories to a given set of sequential histories, called a *sequential specification*. This correctness criterion is based on a relation $\sqsubseteq$ between histories: $h_1 \sqsubseteq h_2$ iff there exists a history h_1' obtained from h_1 by appending return actions that correspond to some of the unmatched call actions in h_1 (completing some pending invocations) and deleting the remaining unmatched call actions in h_1 (removing some pending invocations), such that h_2 is a permutation of h_1' that preserves the order between return and call actions, i.e., if a given return action occurs before a given call action in h_1' then the same holds in h_2. We say that h_2 is a *linearization* of h_1. A history h_1 is called *linearizable* w.r.t. a sequential specification *Seq* iff there exists a sequential history $h_2 \in Seq$ such that $h_1 \sqsubseteq h_2$. An object O is linearizable w.r.t. *Seq* iff each history $h_1 \in H(O)$ is linearizable w.r.t. *Seq*.

Strong linearizability [14] is a strengthening of linearizability which requires that linearizations of an execution can be defined in a prefix-preserving manner. Formally, an object O is *strongly linearizable* w.r.t. *Seq* iff there exists a function $f : E(O) \to Seq$ such that:

1. for any execution $e \in E(O)$, $hist(e) \sqsubseteq f(e)$, and
2. f is prefix-preserving, i.e., for any two executions $e_1, e_2 \in E(O_1)$ such that e_1 is a prefix of e_2, $f(e_1)$ is a prefix of $f(e_2)$.

Strong linearizability has been shown to be equivalent to the existence of a forward simulation (defined below) from O to an *atomic object* $O(Seq)$ defined by the set of sequential histories, *Seq* [5, 27]. Intuitively, if we consider an implementation of a sequential object with histories in *Seq*, then the atomic object $O(Seq)$ corresponds to running the same implementation in a concurrent context provided that method bodies execute in isolation. Formally, the atomic object $O(Seq)$ can be defined as a labeled transition system where:

- the set of states contains pairs formed of a history h and a linearization $h_s \in Seq$ of h, and the initial state contains an empty history and empty linearization,
- the transition labels are call or return actions, or *linearization point* actions $lin(k)$ for linearizing an invocation with identifier k
- the transition relation δ contains all the tuples $((h, h_s), a, (h', h_s'))$, where a is a transition label, such that

$$a \text{ is a call action} \implies h' = h \cdot a \text{ and } h'_s = h_s$$
$$a \text{ is a return action} \implies h' = h \cdot a \text{ and } h'_s = h_s \text{ and } a \text{ occurs in } h'_s$$
$$a = \mathit{lin}(k) \implies h' = h \text{ and } h'_s = h_s \cdot \mathit{call}\ M(x)_k \cdot \mathit{ret}\ y_k, \text{ for some } M,\ x, \text{ and } y.$$

Call actions are only appended to the history h, return actions ensure that the linearization h'_s contains the corresponding method, and linearization point actions extend the linearization with a new method.

The executions of $O(\mathit{Seq})$ are defined as sequences of transitions $s_0, a_0, s_1 \ldots a_{k-1}, s_k$, for some $k > 0$, such that $(s_i, a_i, s_{i+1}) \in \delta$ for each $0 \leq i < k$. Note that $O(\mathit{Seq})$ admits every history which is linearizable w.r.t. *Seq*, i.e., $H(O(\mathit{Seq})) = \{h : \exists h' \in \mathit{Seq}.\ h \sqsubseteq h'\}$.

Given two objects O_1 and O_2, a *forward simulation* from O_1 to O_2 is a (binary) relation F between states of O_1 and O_2 that maps every step of O_1 to a possibly stuttering (no-op) step of O_2. Formally, F is a forward simulation if it contains the pair of initial states of O_1 and O_2, and for every transition (s_1, a, s'_1) of O_1 between two states s_1 and s'_1 with label a and every state s_2 of O_2 such that $(s_1, s_2) \in F$, there exists a state s'_2 of O_2 such that either:

- $s_2 = s'_2$ (stuttering step) and a is not a call or return action, or
- $(s'_1, s'_2) \in F$, (s_2, a', s'_2) is a transition of O_2, and if a is a call or return action, then $a = a'$.

A forward simulation F maps every transition of O_1 starting in a state s_1 to a transition of O_2 which starts in a state s_2 associated by F to s_1. This is different from a related notion of *backward* simulation that maps every transition of O_1 ending in a state s'_1 to a transition of O_2 ending in a state s'_2 associated by the simulation to s'_1 (see [25] for more details).

We say that O_1 *strongly refines* O_2 when there exists a forward simulation from O_1 to O_2. In the context of objects, a generic notion of refinement would correspond to the set of histories of O_1 being included in the set of histories of O_2, which is implied by but not equivalent to the existence of a forward simulation [5, 25]. We may omit the adjective strong for simplicity.

2.2 Message-Passing Implementations

In message-passing implementations, methods can be invoked on a distinguished set of processes called *clients*. Clients are also responsible for returning values of method invocations. The interaction between invocations on different clients may rely on a disjoint set of processes called *servers*. In general, we assume that the processes are asynchronous and communicate by sending and receiving messages that can experience arbitrary delay but are not lost, corrupted, or spuriously generated. Communication is permitted between any pair of processes, not just between clients and servers. A node may run any combination of a client process and a server process. Processes are subject to crash failures; we assume the client process and the server process running on the same node can fail independently, which only strengthens our model.

To simplify the exposition, we model message-passing implementations using labeled transition systems instead of actual code. Each process is defined by a transition system with states in an unspecified set $\mathbb{Q}$. A *message* is a triple $\langle \mathit{src}, \mathit{dst}, v\rangle$ where *src* is the sending process, *dst* is the process to which the message is addressed, and v is the message payload. The set of messages is denoted by $\mathbb{M}\mathit{sgs}$. The transition function δ_j of a server process j is defined as a partial function $\delta_j : \mathbb{Q} \times 2^{\mathbb{M}\mathit{sgs}} \rightharpoonup \mathbb{Q} \times 2^{\mathbb{M}\mathit{sgs}}$. For a given local state s and set of messages *Msgs* received by j, $\delta_j(s, \mathit{Msgs}) = (s', \mathit{Msgs}')$ defines the next local state s' and a set of message Msgs' sent by j. It is possible that *Msgs* or Msgs' is empty. The transition

CALL

$$\frac{i < m \quad s_i = g(i)\downarrow_1 \quad \mathit{pending}_i(s_i) = \mathit{false} \quad \delta_i(s_i, \mathit{call}\ M(x)) = (s_i', \mathit{Msgs})}{g \xrightarrow{\mathit{call}\ M(x)}_i g[i \mapsto \langle s_i', (g(i)\downarrow_2 \cup \mathit{Msgs})\rangle]}$$

RETURN

$$\frac{i < m \quad s_i = g(i)\downarrow_1 \quad \delta_i(s_i, \mathit{ret}\ y) = (s_i', \mathit{Msgs})}{g \xrightarrow{\mathit{ret}\ y}_i g[i \mapsto \langle s_i', (g(i)\downarrow_2 \cup \mathit{Msgs})\rangle]}$$

INTERNAL

$$\frac{s_j = g(j)\downarrow_1 \quad \mathit{Msgs} \subseteq \Big(\bigcup_{0\leq k<m+n} g(k)\downarrow_2\Big)\downarrow_{dst=j} \quad \delta_i(s_j, \mathit{Msgs}) = (s_j', \mathit{Msgs}')}{g \rightarrow_j g[j \mapsto \langle s_j', (g(j)\downarrow_2 \cup \mathit{Msgs}')\rangle]}$$

Figure 1 State transitions of message-passing implementations. We define transitions using a standard notation where the conditions above the line must hold so that the transition given below the line is valid. For a function $f : A \rightharpoonup B$, $f[a \mapsto b]$ denotes the function $f' : A \rightharpoonup B$ defined by $f'(c) = f(c)$, for every $c \neq a$ in the domain of f, and $f'(a) = b$. Also, for a tuple t, $t\downarrow_i$ denotes its i-th component, and for a set of messages *Msgs*, $\mathit{Msgs}\downarrow_{dst=j}$ is the set of messages in *Msgs* with destination j.

function of a client i is defined as $\delta_i : \mathbb{Q} \times (2^{\mathbb{M}sgs} \cup \mathbb{A}) \rightharpoonup \mathbb{Q} \times 2^{\mathbb{M}sgs}$ where $\mathbb{A}$ is a set of call and return actions. Unlike servers, clients are allowed to perform additional *method call* steps or *method return* steps that are determined by call and return actions in $\mathbb{A}$. To simplify the presentation, we assume that a client state records whether an invocation is currently pending and what is the last returned value. Therefore, for a given state s of a client i, $\mathit{pending}_i(s) = \mathit{true}$ iff an invocation is currently pending in state s and $\mathit{retVal}_i(s) = y$ iff there exists a state s' such that $\delta_i(s', \mathit{ret}\ y) = (s, \_)$.

An implementation $I_{mp}(m, n)$ with m client processes and n server processes is defined by an initial local state s_0 that for simplicity, we use to initiate the computation of all processes, and a set $\{\delta_k : 0 \leq k < m + n\}$ of transition functions, where δ_k, $0 \leq k < m$, describe client processes and δ_k, $m \leq k < m + n$, describe server processes.

The executions of a message-passing implementation $I_{mp}(m, n)$ are interleavings of "local" transitions of individual processes. A *global state* g is a function mapping each process to a local state and a pool of messages that the process sent since the beginning of the execution, i.e., $g : [0..m + n - 1] \rightarrow \mathbb{Q} \times 2^{\mathbb{M}sgs}$. The initial global state g_0 maps each process to its initial local state and an empty pool of messages. A transition between two global states advances one process according to its transition function. Figure 1 lists the set of rules defining the transitions of $I_{mp}(m, n)$. CALL and RETURN transition rules correspond to steps of a client due to invoking or returning from a method, and INTERNAL represents steps of a client or a server where it advances its state due to receiving some set of messages. The set of received messages is chosen *non-deterministically* from the pools of messages sent by all the other processes. The non-deterministic choice models arbitrary message delay since it allows sent messages to be ignored in arbitrarily many steps. The messages sent during a step of a process i are added to the pool of messages sent by i and never removed.

An *execution* is a sequence of transition steps $g_0 \rightarrow g_1 \rightarrow \ldots$ between global states. We assume that every message is *eventually delivered*, i.e., for any infinite execution e, a transition step where a process i sends a message *msg* to a process i' can *not* be followed by an infinite set of steps of process i' where the set of received messages in each step excludes *msg*.

▶ Remark 1. For simplicity, our semantics allows a message to be delivered multiple times. We assume that the effects of message duplication can be avoided by including process identifiers and sequence numbers in message payloads. This way a process can track the set of messages it already received from any other process.

We define a notion of crash fault tolerance for message-passing implementations that asks for system-wide progress provided that at most f servers crash. Therefore, an implementation $I_{mp}(m, n)$ is f*-nonblocking* iff for every infinite execution $e = g_0 \rightarrow \ldots \rightarrow g_k \rightarrow \ldots$ and $k > 0$ such that some invocation is pending in g_k, if at least one client and $n - f$ servers execute a step infinitely often in e, then some invocation completes after g_k (i.e., the sequence of transitions in e after g_k includes a RETURN transition).

For m clients and n servers, ABD (as well as its multi-writer version) is f-nonblocking as long as $f < n/2$, while m can be anything. In fact, ABD provides a stronger liveness property, in that every invocation by a non-faulty client eventually completes. Furthermore, ABD only needs client-server communication. So the communication model is weaker that what the model assumes and the output is stronger than what the model requires.

2.3 Shared Memory Implementations

In shared-memory implementations, the code of each method defines a sequence of invocations to a set of *base* objects. In our work, the base objects are standard single-writer (SW) registers. Methods can be invoked in parallel at a number of processes that are asynchronous and crash-prone. We assume that read and write accesses to SW registers are instantaneous.

We omit a detailed formalization of the executions of such an implementation. The pseudo-code we will use to define such implementations can be translated in a straightforward manner to executions seen as sequences of transitions between global states that track values of (local or shared) SW registers and the control point of each process.

We say that a shared-memory implementation is *nonblocking* if for every infinite execution $e = g_0 \rightarrow \ldots \rightarrow g_k \rightarrow \ldots$ and $k > 0$ such that some invocation is pending in g_k, some invocation completes after g_k. The definition of nonblocking for shared-memory implementations demands system-wide progress even if all processes but one fail.

3 Shared-Memory Refinements of Message-Passing Implementations

We show that every message-passing object implementation with m clients and any number n of servers can be refined by a shared-memory implementation with m processes such that: (1) the implementation uses only single-writer registers, and (2) it is nonblocking if the message-passing implementation is $(m - 1)$-nonblocking. By reduction from [18, Corollary 3.7], which shows that there is no nonblocking implementation of several objects, including multi-writer registers, from single-writer registers, the existence of this refinement implies the impossibility of strongly-linearizable message-passing implementations of the same objects no matter how small the fraction of failures is. This reduction relies on the equivalence between strong linearizability and strong refinement and the compositionality of the latter (see Section 4).

The shared-memory implementation should guarantee system-wide progress even if all processes, except one, fail. In contrast, the message-passing implementation only needs to guarantee system-wide progress when no more than f server processes fail. Since the total number of servers may be arbitrarily larger than f, it is impossible to define a "hard-wired" shared-memory refinement where each shared-memory process simulates a pre-assigned message-passing client or server process. Instead, we have each of the m shared-memory processes simulate a client in the message-passing implementation while also cooperating with the other processes in order to simulate steps of all the server processes. This follows the ideas in the BG simulation [7]. Overall, the shared-memory implementation simulates only a subset of the message-passing executions, thereby, it is a *refinement* of the latter. The set of simulated executions is however "complete" in the sense that a method invocation is always enabled on a process that finished executing its last invocation.

The main idea of the refinement is to use a hypothetical message-passing implementation of an object using m clients and n servers as a "subroutine" to implement the object in a system with m processes using SW registers. Each process p in the shared-memory algorithm is associated with a client in the message-passing algorithm, and p, and only p, simulates the steps of that client. Since any number of shared-memory processes may crash, and any number of message-passing clients may crash, this one-to-one association works fine. However, the same approach will not work for simulating the message-passing servers with the shared-memory processes, since the message-passing algorithm might tolerate the failure of only a very small fraction of servers, while the shared-memory algorithm needs to tolerate the failure of all but one of its processes. Instead, all the shared-memory processes cooperate to simulate each of the servers. To this end, each shared-memory process executes a loop in which it simulates a step of its associated client, and then, for each one of the servers in round-robin order, it works on simulating a step of that server. The challenge is synchronizing the attempts by different shared-memory processes to simulate the same step by the same server, without relying on consensus. We use *safe agreement* objects to overcome this difficulty, a separate one for the r-th step of server j, as follows: Each shared-memory process proposes a value, consisting of its local state and a set of messages to send, for the r-th step of server j, and repeatedly checks (in successive iterations of the outer loop) if the value has been resolved, before moving on to the next step of server j. Because of the definition of safe agreement, the only way that server j can be stuck at step r is if one of the simulating shared-memory processes crashes.

The steps of the client and server processes are handled in essentially the same way by a shared memory-memory process, the main difference being that client processes need to react to method invocations and provide responses. The current state of, and set of messages sent by, each message-passing process is stored in a SW register. The shared-memory process reads the appropriate register, uses the message-passing transition function to determine the next state and set of messages to send, and then writes this information into the appropriate register.

More details follow, after we specify safe agreement.

3.1 Safe Agreement Object

The key to the cooperative simulation of server processes is a large set of *safe agreement* objects, each of which is used to agree on a *single* step of a server process. Safe agreement is a weak form of consensus that separates the proposal of a value and the learning of the decision into two methods. A safe agreement object supports two wait-free methods, *propose*, with argument $v \in V$ and return value *done*, and *resolve*, with no argument and return value $v \in V \cup \{\bot\}$. While the methods are both wait-free, *resolve* may return a "non-useful" value $\bot$. Each process using such an object starts with an invocation of *propose*, and continues with a (possibly infinite) sequence of *resolve* invocations; in our simulation, *resolve* is not invoked after it returns a value $v \neq \bot$.

The behavior of a safe agreement object is affected by the possible crash of processes during a method. Therefore, its correctness is not defined using linearizability w.r.t. a sequential specification. Instead, we define such an object to be correct when its (concurrent) histories satisfy the following properties:

Algorithm 1 Method M at process p_i, $0 \leq i < m$. Initially, resolved[j] is true and $r[j]$ is 0, for all $m \leq j < m + n$.

```
Method $M(x)$:
client[$i$] ← ActStep(client[$i$], $call\ M(x)$)                  ▷ simulating the call
while true do
    if $\exists y.\ \delta_i$(client[$i$].state, $ret\ y$) is defined then
        old_client[$i$] ← client[$i$]          ▷ used only to simplify the simulation relation
        client[$i$] ← ActStep(client[$i$], $ret\ y$)                ▷ simulating the return
        return $y$
    end if
    client[$i$] ← InternalStep($i$)                          ▷ simulating a step of client $i$
    for $j \leftarrow m, \ldots, m + n - 1$ do           ▷ simulate at most one step from each server
        if resolved[$j$] then                            ▷ move on to next step of server $j$
            s ← InternalStep($j$)               ▷ returns a new state and pool of sent messages
            r[$j$] ← r[$j$] + 1
            resolved[$j$] ← false
            SA[$j$][r[$j$]].propose(s)
        else                                  ▷ keep trying to resolve current step of server $j$
            s ← SA[$j$][r[$j$]].resolve()
            if s $\neq \perp$ then
                resolved[$j$] ← true
                server[$i$][$j$] ← ⟨s, r[$j$]⟩                   ▷ write to shared SW register
            end if
        end if
    end for
end while
```

- *Agreement:* If two *resolve* methods both return non-⊥ values, then the values are the same.
- *Validity:* The return action of a *resolve* method that returns a value $v \neq \perp$ is preceded by a call action *call propose*(v).
- *Liveness:* If a *resolve* is invoked when there is no pending *propose* method, then it can return only a non-⊥ value.

The liveness condition for safe agreement is weaker than that for consensus, as ⊥ can be returned by *resolve* as long as a *propose* method is pending. Thus it is possible to implement a safe agreement object using SW registers. We present such an algorithm in Appendix A, based on those in [7, 21].

3.2 Details of the Shared-Memory Refinement

Let $I_{mp}(m, n)$ be a message-passing implementation. We define a shared-memory implementation $I_{sm}(m)$ that refines $I_{mp}(m, n)$ and that runs over a set of processes p_i with $0 \leq i < m$. Each process p_i is associated with a client i of $I_{mp}(m, n)$. The code of a method M of $I_{sm}(m)$ executing on a process p_i is listed in Algorithm 1. This code uses the following set of shared objects (the other registers used in the code are local to a process):

- client[i]: SW register written by p_i, holding the current local state (accessed using .state) and pool of sent messages (accessed using .msgs) of client i; $0 \leq i < m$
- server[i][j]: SW register written by p_i, holding the current state and pool of sent messages of server j according to p_i, tagged with a step number (accessed using .sn); $0 \leq i < m$ and $m \leq j < m + n$
- SA[j][r]: safe agreement object used to agree on the r-th step of server j ($m \leq j < m + n$ and $r = 0, 1, \ldots$).

Algorithm 2 Auxiliary functions ACTSTEP, INTERNALSTEP, and COLLECTMESSAGES. MOSTRECENT is a declarative macro used to simplify the code.

```
Function ACTSTEP(client[$i$], $a$):
    return $\langle \delta_i(\text{client}[i].\text{state}, a)\downarrow_1, \text{client}[i].\text{msgs} \cup \delta_i(\text{client}[i].\text{state}, a)\downarrow_2\rangle$

Function INTERNALSTEP($j$) at process $p_i$, $0 \le i < m$:
    Msgs ← COLLECTMESSAGES($j$)
    if $j < m$ then                                              ▷ this is a client process
        ($q$, Msgs') ← $\delta_j$(client[$j$].state, Msgs)       ▷ determine new state and sent messages
        return $\langle q, \text{client}[j].\text{msgs} \cup \text{Msgs'}\rangle$
    else                                                         ▷ this is a server process
        ($q$, Msgs') ← $\delta_j$(server[$i$][$j$].state, Msgs)  ▷ determine new state and sent messages
        return $\langle q, \text{server}[i][j].\text{msgs} \cup \text{Msgs'}\rangle$
    end if

Function COLLECTMESSAGES($j$):
    Msgs ← $\bigcup_{0\le k\le m-1}$ client[$k$].msgs $\downarrow_{dst=j}$   ▷ identify messages sent to $j$ by clients
    for $k \leftarrow m, \ldots, m+n-1$ do                                  ▷ identify messages sent to $j$ by servers
        for $i' \leftarrow 0, \ldots, m-1$ do                               ▷ read the content of server registers
            lserver[$i'$][$k$] ← server[$i'$][$k$]
        end for
        s ← MOSTRECENT(lserver[0..$m-1$][$k$])                              ▷ identify the most recent step of server $k$
        Msgs ← Msgs ∪ s.msgs $\downarrow_{dst=j}$
    end for
    return Msgs
```

MOSTRECENT(lserver[0..m-1][k]) = (lserver[i][k].state, lserver[i][k].msgs) such that lserver[i][k].sn = $max_{0\le j\le m-1}$lserver[j][k].sn

Initially, client[i] stores the initial state and an empty set of messages, for every $0 \le i < m$. Also, server[i][j] stores the initial state, an empty set of messages, and the step number 0, for every $0 \le i < m$ and $m \le j < m + n$.

A process p_i executing a method M simulates the steps that client i would have taken when the same method M is invoked. It stores the current state and pool of sent messages in client[i]. Additionally, it contributes to the simulation of server steps. Each process p_i computes a proposal for the r-th step of a server j (the resulting state and pool of sent messages – see line 11) and uses the safe agreement object SA[j][r] to reach agreement with the other processes (see line 14). It computes a proposal for a next step of server j only when agreement on the r-th step has been reached, i.e., it gets a non-$\perp$ answer from SA[j][r].resolve() (see the if conditions at lines 10 and 17). However, it can continue proposing or agreeing on steps of other servers. It iterates over all server processes in a round-robin fashion, going from one server to another when resolve() returns $\perp$. This is important to satisfy the desired progress guarantees.

Steps of client or server processes are computed locally using the transition functions of $I_{mp}(m, n)$ in ACTSTEP and INTERNALSTEP, listed in Algorithm 2. A method M on a process p_i starts by advancing the state of client i by simulating a transition labeled by a call action (line 1). To simulate an "internal" step of client i (or a server step), a subtle point is computing the set of messages that are supposed to be received in this step. This is done by reading all the registers client[_] and server[_][_] in a sequence and collecting the set of messages in client[_].msgs or server[_][_].msgs that have i as a destination. Since the shared-memory processes can be arbitrarily slow or fast in proposing or observing agreement on the steps of a server j, messages are collected only from the "fastest" process, i.e., the process p_k such that server$[k][j]$ contains the largest step number among server$[0..m\text{-}1][j]$ (see the mostRecent macro). This is important to ensure that messages are eventually delivered. Since the set of received messages contains *all* the messages from client[_].msgs or server[_][_].msgs with destination i as opposed to a non-deterministically chosen subset (as in the semantics of $I_{mp}(m,n)$ – see Figure 1), some steps of $I_{mp}(m,n)$ may not get simulated by this shared-memory implementation. However, this is not required as long as the shared-memory implementation allows methods to be invoked arbitrarily on "idle" processes (that are not in the middle of another invocation). This is guaranteed by the fact that each client is simulated locally by a different shared-memory process. A process p_i returns whenever a return action is enabled in the current state stored in client$[i]$ (see the condition at line 3). Server steps are computed in a similar manner to "internal" steps of a client.

3.3 Correctness of the Shared-Memory Refinement

We prove that there exists a forward simulation from the shared-memory implementation defined in Algorithm 1 to the underlying message-passing implementation $I_{mp}(m,n)$, which proves that the former is a (strong) *refinement* of the latter. The proof shows that roughly, the message passing state defined by the content of all registers client$[i]$ with $0 \leq i < m$ and the content of all registers server$[i][j]$ that have the highest step number among server$[i'][j]$ with $0 \leq i' < m$ is reachable in $I_{mp}(m,n)$. Each write to a register client$[i]$ corresponds to a transition in the message-passing implementation that advances the state of client i, and each write to server$[i][j]$ containing a step number that is written for the first time among all writes to server[_][j] corresponds to a transition that advances the state of server j. This choice is justified since the same value is written in these writes, by properties of safe agreement. Then, we also prove that $I_{sm}(m)$ is nonblocking provided that $I_{mp}(m,n)$ is $(m-1)$-nonblocking.

▶ **Theorem 2.** *$I_{sm}(m)$ is a refinement of $I_{mp}(m,n)$.*

Proof. We define a relation F between shared-memory and message-passing global states as follows: every global state of Algorithm 1 is associated by F with a message-passing global state g such that for every client process $0 \leq i < m-1$ and server process $m \leq j < n$,

$$g(i) = \begin{cases} \text{actStep}(\text{client}[i], \textit{call } M(x)), & \text{if } p_i \text{ is before control point 2 in Algorithm 1} \\ \text{old\_client}[i], & \text{if } p_i \text{ is at control points 5 or 6 in Algorithm 1} \\ \text{client}[i], & \text{otherwise} \end{cases}$$

$$g(j) = \text{mostRecent}(\text{server}[0..m-1][j])$$

The first two cases in the definition of $g(i)$ are required so that call and return transitions in shared-memory are correctly mapped to call and return transitions in message-passing. The first case concerns call transitions and intuitively, it provides the illusion that a shared-memory call and the first statement in the method body (at line 1) are executed instantaneously at the same time. The second case concerns return transitions and "delays" the last statement before return (at line 5) so that it is executed instantaneously with the return.

Note that F is actually a function since the message-passing global state is uniquely determined by the process control points and the values of the registers in the shared-memory global state. Also, the use of mostRecent is well defined because server$[i][j]$.sn = server$[i'][j]$.sn implies that server$[i][j]$.state = server$[i'][j]$.state and server$[i][j]$.msgs = server$[i'][j]$.msgs, for every $0 \leq i, i' < m$ (due to the use of the safe agreement objects).

In the following, we show that F is indeed a forward simulation. Let us consider an indivisible step of Algorithm 1 going from a global state v_1 to a global state v_2, and g_1 the message-passing global state associated with v_1 by F. We show that going from g_1 to the message-passing global state g_2 associated with v_2 by F is a valid (possibly stuttering) step of the message-passing implementation. We also show that call and return steps of Algorithm 1 are simulated by call and return steps of the message-passing implementation, respectively.

We start the proof with call and return steps. Thus, consider a step of Algorithm 1 going from v_1 to v_2 by invoking a method M with argument x on a process p_i. Invoking a method in Algorithm 1 will only modify the control point of p_i. Therefore, the message-passing global states g_1 and g_2 differ only with respect to process i: $g_1(i)$ is the value of client[i] in v_1 while $g_2(i)$ is the result of ACTSTEP on that value and *call* $M(x)$ (since the process is before control point 2). Therefore, $g_1 \xrightarrow{call\ M(x)}_i g_2$ (cf. Figure 1). For return steps of Algorithm 1, g_1 and g_2 also differ only with respect to process i: $g_1(i)$ is the value of old_client[i] in v_1 (since the process is at control point 6) while $g_2(i)$ is the value of client[i] in v_2. From lines 4–6 of Algorithm 1, we get that the value of client[i] in v_2 equals the value of ACTSTEP for old_client[i] in v_1 and the action *ret* y (note that old_client[i] and client[i] are updated only by the process p_i). Therefore, $g_1 \xrightarrow{ret\ y}_i g_2$ (cf. Figure 1).

Every step of Algorithm 1 except for the writes to client[i] or server[i][j] at lines 8 and 19 is mapped to a stuttering step of the message-passing implementation. This holds because F associates the same message-passing global state to the shared-memory global states before and after such a step.

Let us consider a step of Algorithm 1 executing the write to client[i] at line 8 (we refer to the write that happens once INTERNALSTEP(i) has finished – we do *not* assume that line 8 happens instantaneously). We show that it is simulated by a step of client i of the message-passing implementation. By the definition of INTERNALSTEP, the value of client[i] in v_2 is obtained by applying the transition function of process i on the state stored in client[i] of v_1 and some set of messages *Msgs* collected from client[i'] and server[i'][j] with $0 \leq i' < m$ and $m \leq j < n$. *Msgs* is computed using the function COLLECTMESSAGES that reads values of client[i'] and server[i'][j] in shared-memory states that may precede v_1. However, since the set of messages stored in each of these registers increases monotonically[1], *Msgs* is included in the set of messages stored in v_1 (i.e., the union of client[i'].msgs and server[i'][j].msgs for all $0 \leq i' < m$ and $m \leq j < n$). Therefore,

$$Msgs \subseteq \Big(\bigcup_{0 \leq k < n} g_1(k) \downarrow_2 \Big) \downarrow_{dst=j},$$

which together with the straightforward application of δ_i in INTERNALSTEP implies that $g_1 \rightarrow_i g_2$.

Finally, let us consider a step of Algorithm 1 executing the write to server[i][j] at line 19. Let $\langle s, t \rangle$ be the value written to server[i][j] in this step. If there exists some other process $p_{i'}$ such that the register server[i'][j] in v_1 stores a tuple $\langle s', t' \rangle$ with $t \leq t'$, then this step is mapped to a stuttering step of the message-passing implementation. Indeed, the use of MOSTRECENT in the definition of F implies that it associates the same message-passing global state to the shared-memory states before and after such a step. Otherwise, we show that this write is simulated by a step of server j of the message-passing implementation. By the specification of the safe agreement objects, s is a proposed value,

[1] This is a straightforward inductive invariant of Algorithm 1.

and therefore, computed using INTERNALSTEP by a possibly different process $p_{i'}$. During this INTERNALSTEP computation server$[i'][j]$ stores a value of the form $\langle s', t-1\rangle$, for some s' (cf. the increment at line 12). Since the values stored in the server$[i'][j]$ registers are monotonic w.r.t. their step number component, it must be the case that s' is the outcome of MOSTRECENT(server$[0..m-1][j]$) when applied on the global state v_1. Therefore, the INTERNALSTEP computation of $p_{i'}$ applies δ_j on the state $g_1(j)\downarrow_1$ and a set of messages *Msgs* computed using COLLECTMESSAGES. As in the case of the client$[i]$ writes,

$$Msgs \subseteq (\bigcup_{0\leq k<n} g_1(k)\downarrow_2)\downarrow_{dst=j},$$

which implies that $g_1 \rightarrow_j g_2$. ◀

The message-passing executions simulated by the shared-memory executions satisfy the eventual message delivery assumption. Indeed, since all the shared objects are wait-free, a message *msg* stored in client$[i]$ or server$[i][j]$ will be read by all non-failed processes in a finite number of steps. Therefore, if *msg* is sent to a client process i', then it will occur in the output of INTERNALSTEP(i') at line 8 on process $p_{i'}$ after a finite number of invocations of this function. Also, if *msg* is sent to a server process j', then it will be contained in the output of INTERNALSTEP(j') at line 11 on every non-failed process $p_{i'}$ with $0 \leq i' < m$ after a finite number of steps.

In the following, we show that the shared-memory implementation is nonblocking (guarantees system-wide progress for m processes, any number of which can fail) assuming that the message-passing implementation guarantees system-wide progress if at most $m-1$ servers fail.

▶ **Theorem 3.** *If $I_{mp}(m,n)$ is $(m-1)$-nonblocking, then $I_{sm}(m)$ is nonblocking.*

Proof. Since Algorithm 1 uses only wait-free objects (SW registers and safe agreement objects), an invocation of a method M at a non-crashed process could be non-terminating only because the *resolve* invocations on safe agreement objects return $\perp$ indefinitely. The latter could forbid the progress of a single server process. By the specification of safe agreement, *resolve* can return $\perp$ only if it started while a *propose* invocation (on the same object) is pending. Since a process p_i has at most one invocation of *propose* pending at a time, the number of *propose* invocations that remain unfinished indefinitely is bounded by the number of failed shared-memory processes. Therefore, $m-1$ failed shared-memory processes forbid progress on at most $m-1$ server processes. Since, $I_{mp}(m,n)$ is $(m-1)$-nonblocking, we get that $I_{sm}(m)$ is nonblocking. ◀

The proof above also applies to an extension of Theorem 3 to wait-freedom, i.e., $I_{sm}(m)$ is wait-free if $I_{mp}(m,n)$ ensures progress of individual clients assuming at most $m-1$ server failures.

4 Impossibility Results

We show the impossibility of strongly-linearizable nonblocking implementations in an asynchronous message-passing system for several highly useful objects (including multi-writer registers). This impossibility result is essentially a reduction from [18, Corollary 3.7] that states a corresponding result for shared-memory systems. Since strong linearizability and (strong) refinement are equivalent and refinement is compositional [5, 25, 27], the results in Section 3 imply that any strongly-linearizable message-passing implementation can be used to define a strongly-linearizable implementation in shared-memory. Since the latter also preserves the nonblocking property, the existence of a message-passing implementation would contradict the shared-memory impossibility result.

▶ **Theorem 4.** *Given a sequential specification Seq, there is a nonblocking shared-memory implementation with m processes, which is strongly linearizable w.r.t. Seq and which only uses SW registers, if there is a nonblocking message-passing implementation with m clients and an arbitrary number n of servers, which is strongly linearizable w.r.t. Seq.*

Proof. Given a message-passing implementation $I_{mp}(m, n)$ as above, Theorem 2 and Theorem 3 show that the shared-memory implementation $I_{sm}(m)$ defined in Algorithm 1 is a refinement of $I_{mp}(m, n)$ and nonblocking. Since strong linearizability w.r.t. *Seq* is equivalent to refining $O(\mathit{Seq})$ (see Section 2) and the refinement relation (defined by forward simulations) is transitive[2], we get that $I_{sm}(m, n)$ is a refinement of $O(\mathit{Seq})$, which implies that it is strongly linearizable w.r.t. *Seq*. Finally, Theorem 6 shows that the safe agreement objects in $I_{sm}(m)$ can be implemented only using SW registers, which implies that $I_{sm}(m)$ only relies on SW registers. ◀

▶ **Corollary 5.** *There is no strongly linearizable nonblocking message-passing implementation with three or more clients of multi-writer registers, max-registers, counters, or snapshot objects.*

Proof. If such an implementation existed, then Theorem 4 would imply the existence of a strongly linearizable nonblocking implementation from single-writer registers, which is impossible by [18, Corollary 3.7]. ◀

5 Conclusions and Related Work

In order to exploit composition and abstraction in message-passing systems, it is crucial to understand how properties of randomized programs are preserved when they are composed with object implementations. This paper extends the study of strong linearizability to message-passing object implementations, showing how results for shared-memory object implementations can be translated. Consequently, there can be no strongly-linearizable crash-tolerant message-passing implementations of multi-writer registers, max-registers, counters, or snapshot objects.

In the context of shared-memory object implementations, several results have shown the limitations of strongly-linearizable implementations. Nontrivial objects, including multi-writer registers, max registers, snapshots, and counters, have no nonblocking strongly-linearizable implementations from single-writer registers [18]. In fact, even with multi-writer registers, there is no wait-free strongly-linearizable implementation of a monotonic counter [9], and, by reduction, neither of snapshots nor of max-registers. Queues and stacks do not have an n-process nonblocking strongly-linearizable implementation from objects whose readable versions have consensus number less than n [4].

On the positive side, any consensus object is strongly linearizable, which gives an *obstruction-free* strongly-linearizable universal implementation (of any object) from single-writer registers [18]. Helmi et al. [18] also give a *wait-free* strongly-linearizable implementation

[2] If there is a forward simulation F_1 from O_1 to O_2 and a forward simulation F_2 from O_2 to O_3, then the composition $F_1 \circ F_2 = \{(s_1, s_3) : \exists s_2.(s_1, s_2) \in F_1 \wedge (s_2, s_3) \in F_2\}$ is a forward simulation from O_1 to O_3.

of *bounded* max register from multi-writer registers [18]. When updates are strongly linearizable, objects have *nonblocking* strongly-linearizable implementations from multi-writer registers [9]. The space requirements of the latter implementation is avoided in a *nonblocking* strongly-linearizable implementation of snapshots [26]. This snapshot implementation is then employed with an algorithm of [2] to get a *nonblocking* strongly-linearizable universal implementation of any object in which all methods either commute or overwrite.

The *BG simulation* has been used in many situations and several communication models. Originally introduced for the shared-memory model [7], it showed that t-fault-tolerant algorithms to solve *colorless* tasks (like set agreement) among n processes, can be translated into t-fault-tolerant algorithms for $t+1$ processes (i.e., wait-free algorithms) for the same problem. The *extended* BG simulation [12] also works for so-called *colored* tasks, where different processes must decide on different values. Another extension of the BG simulation [11] was used to dynamically reduce synchrony of a system. (See additional exposition in [21, 23].)

To the best of our knowledge, all these simulations allow only a single invocation by each process, and none of them handles *long-lived* objects. Furthermore, they are either among different variants of the shared-memory model [7, 11, 12, 23] or among different failure modes in the message-passing model [10, 22].

This paper deals with multi-writer registers and leaves open the question of finding a strongly-linearizable message-passing implementation of a *single-writer* register. The original ABD register implementation [3], which is for a single writer, is not strongly linearizable [16].

Recently, two ways of mitigating the bad news of this paper have been proposed, both of which move away from strong linearizability. In [17], a consistency condition that is intermediate between linearizability and strong linearizability, called "write strong-linearizability" is defined and it is shown that for some program this condition is sufficient to preserve the property of having non-zero termination probability, and that a variant of ABD satisfies write strong-linearizability. In another direction, [6] presents a simple modification to ABD that preserves the property of having non-zero termination probability; the modification is to query the servers multiple times instead of just once and then randomly pick which set of responses to use. This modification also applies to the snapshot implementation in [1]; note that snapshots do not have nonblocking strongly-linearizable implementations, in either shared-memory (proved in [18]) or message-passing (as we prove in this paper, by reduction).

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

2 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *SPAA*, pages 340–349, 1990.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

4 Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. *Journal of Parallel and Distributed Computing*, 121:1–14, 2018.

5 Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In *DISC*, pages 2:1–2:17, 2019.

6 Hagit Attiya, Constantin Enea, and Jennifer L. Welch. Linearizable implementations suffice for termination of randomized concurrent programs. *CoRR*, abs/2106.15554, 2021. `arXiv:2106.15554`.

7 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.

8 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

9 Oksana Denysyuk and Philipp Woelfel. Wait-freedom is harder than lock-freedom under strong linearizability. In *DISC*, pages 60–74, 2015.

10 Danny Dolev and Eli Gafni. Synchronous hybrid message-adversary. *CoRR*, abs/1605.02279, 2016. arXiv:1605.02279.

11 Pierre Fraigniaud, Eli Gafni, Sergio Rajsbaum, and Matthieu Roy. Automatically adjusting concurrency to the level of synchrony. In *DISC*, pages 1–15, 2014.

12 Eli Gafni. The extended BG-simulation and the characterization of t-resiliency. In *STOC*, pages 85–92, 2009.

13 Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

14 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, pages 373–382, 2011.

15 Theophanis Hadjistasi, Nicolas Nicolaou, and Alexander A. Schwarzmann. Oh-ram! one and a half round atomic memory. In *NETYS*, pages 117–132. Springer International Publishing, 2017.

16 Vassos Hadzilacos, Xing Hu, and Sam Toueg. On atomic registers and randomized consensus in M&M systems (version 4). *CoRR*, abs/1906.00298, 2020. arXiv:1906.00298.

17 Vassos Hadzilacos, Xing Hu, and Sam Toueg. On register linearizability and termination. In *PODC*, pages 521–531, 2021.

18 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *PODC*, pages 385–394, 2012.

19 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

20 Kaile Huang, Yu Huang, and Hengfeng Wei. Fine-grained analysis on fast implementations of distributed multi-writer atomic registers. In *PODC*, pages 200–209, 2020.

21 Damien Imbs and Michel Raynal. Visiting Gafni's reduction land: From the BG simulation to the extended BG simulation. In *SSS*, pages 369–383, 2009.

22 Damien Imbs, Michel Raynal, and Julien Stainer. Are Byzantine failures really different from crash failures? In *DISC*, pages 215–229, 2016.

23 Petr Kuznetsov. Universal model simulation: BG and extended BG as examples. In *SSS*, pages 17–31, 2013.

24 Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *FTCS*, pages 272–281, 1997.

25 Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

26 Sean Ovens and Philipp Woelfel. Strongly linearizable implementations of snapshots and other types. In *PODC*, pages 197–206, 2019.

27 Amgad Sadek Rady. Characterizing Implementations that Preserve Properties of Concurrent Randomized Algorithms. Master's thesis, York University, Toronto, Canada, 2017.

A An Implementation of Safe Agreement

We present an algorithm to implement a safe agreement object that only uses *single-writer* registers. The algorithm is based on [7, 21].

The crux of the safe agreement algorithm is to identify a *core* set of processes, roughly, those who were first to start the algorithm. Once the core set is identified, the proposal of a fixed process in this set is returned. Our algorithm picks the proposal of the process with minimal id, but the process with maximal id can be used just as well. A "double collect" mechanism is used to identify the core set, by having every process write its id and repeatedly

Algorithm 3 Safe agreement, code for process p_i.

```
1:  Propose($v$):
2:  $Val[i] \leftarrow v$                                            ▷ announce own proposal
3:  $Id[i] \leftarrow i$                                          ▷ announce own participation
4:  repeat                                                          ▷ double collect
5:      for $j \leftarrow 0, \ldots, m-1$ do $collect1[j] \leftarrow Id[j]$
6:      for $j \leftarrow 0, \ldots, m-1$ do $collect2[j] \leftarrow Id[j]$
7:  until $collect1 = collect2$                              ▷ all components are equal
8:  $Set[i] \leftarrow \{j : collect1[j] \neq \bot\}$

9:  Resolve():
10: for $j \leftarrow 0, \ldots, m-1$ do $s[j] \leftarrow Set[j]$                ▷ read $m$ registers
11: $C \leftarrow$ smallest (by containment) non-empty set in $s[0, \ldots, m-1]$
12: if for every $j \in C$, $((s[j] \neq \emptyset)$ and $(C \subseteq s[j]))$ then
13:     return $Val[\min(C)]$        ▷ the proposal of the process with minimal id in $C$
14: else
15:     return $\bot$                                                ▷ no decision yet
16: end if
```

read all the processes' corresponding variables until it observes no change.[3] The process then writes the set consisting of all the ids collected. To resolve, a process reads all these sets, and intuitively, wishes to take the smallest set among them, C, as the core set. However, it is possible that an even smaller set will be written later. The key insight of the algorithm (identified by [7]) is that such a smaller set can only be written by a process whose identifier is already in C. Thus, once all processes in C wrote their sets, either one of them is strictly contained in C (and hence, can replace it), or no smaller set will ever be written.

The pseudocode is listed in Algorithm 3. The algorithm uses the following single-writer shared registers (the other registers used in the code are local to a process):

- $Val[i]$: register written by p_i, holding a proposal, initially $\bot$; $0 \leq i < m$
- $Id[i]$: register written by p_i, holding its own id; initially $\bot$; $0 \leq i < m$
- $Set[i]$: register written by p_i, holding a set of process ids, initially $\emptyset$; $0 \leq i < m$

Notice that *propose* and *resolve* are wait-free. This is immediate for *resolve*. For *propose*, note that the double collect loop (in Lines 4–7) is executed at most m times, since there are at most m writes to Id (one by each process).

▶ **Theorem 6.** *Algorithm 3 is an implementation of safe agreement from single-writer registers.*

Proof. To show validity, first note that a non-$\bot$ value v returned by any *resolve* method is that stored in $Val[i]$ for some i such that p_i wrote to its $Id[i]$ shared variable. The code ensures that before p_i writes to $Id[i]$, it has already written v to $Val[i]$, in response to the invocation of *propose*(v).

Agreement and liveness hinge on the following comparability property of the sets of ids written to the array Set:

[3] This use of double collect is a stripped-down version of the snapshot algorithm [1].

▶ **Lemma 7.** *For any two processes p_i and p_j, if p_i writes S_i to* $\text{Set}[i]$ *and p_j writes S_j to* $\text{Set}[j]$*, then either $S_i \subseteq S_j$ or $S_j \subseteq S_i$.*

Proof. Assume by contradiction that S_i and S_j are incomparable, i.e., there exist $i' \in S_i \setminus S_j$ and $j' \in S_j \setminus S_i$. Without loss of generality, let us assume that $p_{i'}$ writes its id to $Id[i']$ before $p_{j'}$ writes its id to $Id[j']$ (otherwise, a symmetric argument applies). Since $j' \in S_j$, the last collect in the loop at line 4 on process p_j starts after $p_{j'}$ writes to $Id[j']$ and $p_{i'}$ writes to $Id[i']$. Therefore, the process p_j must have read i' from $Id[i']$ in this collect (i.e., $collect2[i'] = i'$), which contradicts the assumption that $i' \notin S_j$. ◀

Suppose p_i returns a non-$\bot$ value $Val[k]$ because k is the smallest id in $C = s[h]$, which is the smallest Set read by p_i in Line 10, and $p_{i'}$ returns a non-$\bot$ value $Val[k']$ because k' is the smallest id in $C' = s[h']$, which is the smallest Set read by $p_{i'}$ in Line 10. Assume in contradiction that $k \neq k'$, which implies that $C \neq C'$ and $h \neq h'$. By Lemma 7, C and C' are comparable; without loss of generality, assume $C \subseteq C'$. Then $C \subset C'$, which contradicts the condition for returning a non-$\bot$ value (Line 12) in $p_{i'}$. Indeed, since $h \in C \subset C'$ (every process reads *Id* registers after writing to its own), $p_{i'}$ should have read $Set[h]$ before returning and witnessed the fact that it contains a smaller set than $Set[h']$.

We now consider liveness. Assume no process has an unfinished *propose* method. Thus, every process that writes to its *Id* variable in Line 3, also writes to its *Set* variable in Line 8. Consider any *resolve* method, say by p_i, that begins after the last *propose* method completes. Let C be the smallest non-empty set obtained by p_i in Line 10. For each $j \in C$, $Set[j]$ is not empty, since all the *propose* methods completed. By the choice of C, Lemma 7 ensures that C is a subset of $Set[j]$. Thus p_i returns a non-$\bot$ value in Line 13. ◀

Locally Checkable Labelings with Small Messages

Alkida Balliu
Universität Freiburg, Germany

Keren Censor-Hillel
Technion, Haifa, Israel

Yannic Maus
Technion, Haifa, Israel

Dennis Olivetti
Universität Freiburg, Germany

Jukka Suomela
Aalto University, Finland

Abstract

A rich line of work has been addressing the computational complexity of locally checkable labelings (LCLs), illustrating the landscape of possible complexities. In this paper, we study the landscape of LCL complexities under bandwidth restrictions. Our main results are twofold. First, we show that on trees, the CONGEST complexity of an LCL problem is asymptotically equal to its complexity in the LOCAL model. An analog statement for non-LCL problems is known to be false. Second, we show that for general graphs this equivalence does not hold, by providing an LCL problem for which we show that it can be solved in $O(\log n)$ rounds in the LOCAL model, but requires $\tilde{\Omega}(n^{1/2})$ rounds in the CONGEST model.

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, CONGEST, locally checkable labelings

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.8

Related Version *Full Version*: https://arxiv.org/abs/2105.05574

Funding This project was partially supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 755839 (Keren Censor-Hillel, Yannic Maus).

1 Introduction

Two standard models of computing that have been already used for decades to study distributed graph algorithms are the LOCAL model and the CONGEST model [36]. In the LOCAL model, each node in the network can send *arbitrarily large messages* to each neighbor in each round, while in the CONGEST model the nodes can only send *small messages* (we will define the models in Section 2). In general, being able to send arbitrarily large messages can help a lot: there are graph problems that are trivial to solve in the LOCAL model and very challenging in the CONGEST model, and this also holds in trees.

Nevertheless, we show that there is a broad family of graph problems – *locally checkable labelings* or LCLs in short – in which the two models of computing have exactly the same expressive power *in trees* (up to constant factors): if a locally checkable labeling problem Π can be solved in trees in $T(n)$ communication rounds in the LOCAL model, it can be solved in $O(T(n))$ rounds also in the CONGEST model. We also show that this is no longer the case if we switch from trees to general graphs:

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 8; pp. 8:1–8:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

	LCL *problems (our work)*	*General problems (prior work)*
Trees:	CONGEST = LOCAL	CONGEST ≠ LOCAL
General graphs:	CONGEST ≠ LOCAL	CONGEST ≠ LOCAL

Locally Checkable Labelings. The study of the distributed computational complexity of locally checkable labelings (LCLs) in the LOCAL model was initiated by Naor and Stockmeyer [34] in the 1990s, but this line of research really took off only in the recent years [3–12, 14, 15, 17, 21–23, 35].

LCLs are a family of graph problems: an LCL problem Π is defined by listing a *finite set of valid labeled local neighborhoods*. This means that Π is defined on graphs of some finite maximum degree Δ, and the task is to label the vertices and/or edges with labels from some finite set so that the labeling satisfies some set of local constraints (see Section 2 for the precise definition).

A simple example of an LCL problem is the task of coloring a graph of maximum degree Δ with $\Delta+1$ colors (here valid local neighborhoods are all properly colored local neighborhoods). LCLs are a broad family of problems, and they contain many key problems studied in the field of distributed graph algorithms, including graph coloring, maximal independent set and maximal matching.

Classification of LCL problems. One of the key questions related to the LOCAL model has been this: given an arbitrary LCL problem Π, what can we say about its computational complexity in the LOCAL model (i.e., how many rounds are needed to solve Π)? It turns out that we can say quite a lot. There are infinitely many distinct complexity classes, but there are also some wide *gaps* between the classes – for example, if Π can be solved with a deterministic algorithm in $o(\log n)$ rounds, it can also be solved in $O(\log^* n)$ rounds [22].

Furthermore, some parts of the classification are *decidable*: for example, in the case of rooted trees, we can feed the description of Π to a computer program that can determine the complexity of Π in the LOCAL model [8]. The left part of Figure 1 gives a glimpse of what is known about the landscape of possible complexities of LCL problems in the LOCAL model.

However, this entire line of research has been largely confined to the LOCAL model. Little is known about the general structure of the landscape of LCL problems in the CONGEST model. In some simple settings (in particular, paths, cycles, and rooted trees) it is known that the complexity classes are the same between the two models [8, 24], but this has been a straightforward byproduct of work that has aimed at classifying the problems in the LOCAL model. What happens in the more general case has been wide open – the most interesting case for us is LCL problems in (unrooted) trees.

Prior work on LCLs in trees. In the case of trees, LCL problems are known to exhibit a broad variety of different complexities in the LOCAL model. For example, for every $k = 1, 2, 3, \ldots$ there exists an LCL problem whose complexity in the LOCAL model is exactly $\Theta(n^{1/k})$ [23]. There are also problems in which randomness helps exponentially: for example, the sinkless orientation problem belongs to the class of problems that requires $\Theta(\log n)$ rounds for deterministic algorithms and only $\Theta(\log \log n)$ rounds for randomized algorithms in the LOCAL model [17, 22]. Until very recently, one open question related to LCLs in trees remained: whether there are any problems in the region between $\omega(1)$ and $o(\log^* n)$; there is now a (currently unpublished) result [16] that shows that no such problems exist, and this completed the classification of LCLs in trees in the LOCAL model.

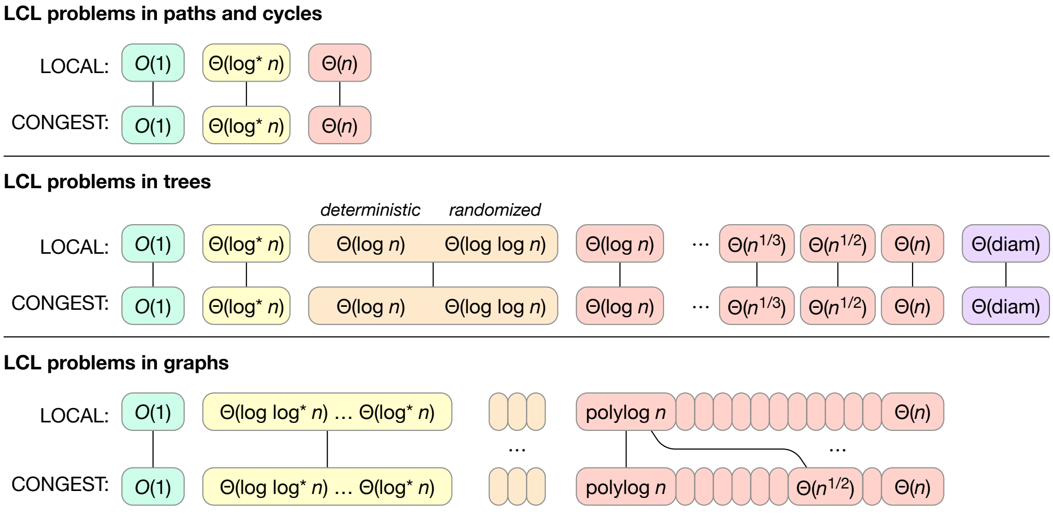


Figure 1 The landscape of LCL problems in the LOCAL and CONGEST models.

Prior work separating CONGEST and LOCAL. In the LOCAL model, all natural graph problems can be trivially solved in $O(n)$ rounds and also in $O(D)$ rounds, where D is the diameter of the input graph: in $O(D)$ rounds all nodes can gather the full information on the entire input graph.

However, there are many natural problems that do not admit $O(D)$-round algorithms in the CONGEST model. Some of the best-known examples include the task of finding an (approximate) minimum spanning tree, which requires $\tilde{\Omega}(\sqrt{n} + D)$ rounds [26, 37, 38], and the task of computing the diameter, which requires $\tilde{\Omega}(n)$ rounds [1, 30]. There are also natural problems that do not even admit $O(n)$-round algorithms in the CONGEST model. For example, finding an exact minimum vertex cover or dominating set requires $\tilde{\Omega}(n^2)$ rounds [2, 19].

Moreover, separations also hold in some cases where the LOCAL complexity is constant: One family of such problems is that of detecting subgraphs, for which an extreme example is that for any k there exists a subgraph of diameter 3 and size $O(k)$, which requires $\Omega(n^{2-1/k})$ rounds to detect in CONGEST, even when the network diameter is also 3 [29]. Another example is spanner approximations, for which there is a constant-round $O(n^\epsilon)$-approximation algorithm in the LOCAL model [13], but $\tilde{\Omega}(n^{1/2-\epsilon/2})$ rounds are needed in the CONGEST model (and even $\tilde{\Omega}(n^{1-\epsilon/2})$ rounds for deterministic algorithms) [18]. Separations hold also in trees: all-pairs shortest-paths on a star can be solved in 2 LOCAL rounds, but requires $\tilde{\Omega}(n)$ CONGEST rounds [19, 33].

While these lower bound results do not have direct implications in the context of LCL problems, they show that there are many natural settings in which the LOCAL model is much stronger than the CONGEST model.

1.1 Our Contributions

We show that

LOCAL = CONGEST for LCL problems in trees.

Not only do we have the same round complexity classes, but every LCL problem has the same asymptotic complexity in the two models. In particular, our result implies that *all* prior results related to LCLs in trees hold also in the CONGEST model. For example all decidability results for LCLs in trees in the LOCAL model hold also in the CONGEST model; this includes the decidable gaps in [23] and [21]. We also show that the equivalence holds not only if we study the complexity as a function of n, but also for problems with complexity $\Theta(D)$.

Given the above equality, one could conjecture that the LOCAL model does not have any advantage over the CONGEST model for any LCL problem. We show that this is not the case: as soon as we step outside the family of trees, we can construct an example of an LCL problem that is solvable in $\operatorname{polylog} n$ rounds in the LOCAL model but requires $\Omega(\sqrt{n})$ rounds in the CONGEST model. In summary, we show that

LOCAL $\neq$ CONGEST for LCL problems in general graphs.

Here "general graphs" refers to general *bounded-degree* graphs, as each LCL problem comes with some finite maximum degree Δ. We summarize our main results in Figure 1.

Open questions. The main open question after the present work is how wide the gap between CONGEST and LOCAL can be made in general graphs. More concretely, it is an open question whether there exists an LCL problem that is solvable in $O(\log n)$ rounds in the LOCAL model but requires $\Omega(n)$ rounds in the CONGEST model.

1.2 Road Map, Key Techniques, and New Ideas

We prove the equivalence of LOCAL and CONGEST in trees in Sections 3–5, and we show the separation between LOCAL and CONGEST in general graphs in Section 6. We start in Section 3 with some basic facts in the $O(\log n)$ regime that directly follow from prior work. The key new ideas are in Sections 4–6.

Equivalence in trees: superlogarithmic region (Section 4). The first major challenge is to prove that any LCL with some sufficiently high complexity $T(n)$ in the LOCAL model in trees has exactly the same asymptotic complexity $\Theta(T(n))$ also in the CONGEST model. A natural idea would be to show that a given LOCAL model algorithm A can be simulated efficiently in the CONGEST model. However, this is *not possible* in general – the proof has to rely somehow on the property that A solves an LCL problem.

Instead of a direct simulation approach, we use as a starting point prior *gap results* related to LCLs in the LOCAL model. A typical gap result in trees can be phrased as follows, for some $T_2 \ll T_1$:

- Given: a $T_1(n)$-round algorithm A_1 that solves some LCL Π in trees in the LOCAL model.
- We can construct: a $T_2(n)$-round algorithm A_2 that solves it in the LOCAL model.

We *amplify* the gap results so that we arrive at theorems of the following form – note that we not only speed up algorithms but also switch to a weaker model:

- Given: a $T_1(n)$-round algorithm A_1 that solves some LCL Π in trees in the **LOCAL** model.
- We can construct: a $T_2(n)$-round algorithm A_2' that solves it in the **CONGEST** model.

At first, the entire approach may seem hopeless: clearly A_2' has to somehow depend on A_1, but how could a fast CONGEST-model algorithm A_2' possibly make any use of a slow LOCAL-model algorithm A_1 as a black box? Any attempts of simulating (even a small number of rounds) of A_1 seem to be doomed, as A_1 might use large messages.

We build on the strategy that Chang and Pettie [23] and Chang [21] developed in the context of the LOCAL model. In their proofs, A_2 does not make direct use of A_1 as a black box, but mere *existence* of A_1 guarantees that Π is sufficiently well-behaved in the sense that long path-like structures can be labeled without long-distance coordination.

This observation served as a starting point in [21, 23] for the development of an efficient LOCAL model algorithm A_2 that finds a solution that is possibly different from the solution produced by A_1, but it nevertheless satisfies the constraints of problem Π. However, A_2 obtained with this strategy abuses the full power of large messages in the LOCAL model.

Hence while our aim is at understanding the landscape of computational complexity, we arrive at a *concrete algorithm design challenge*: we need to design a CONGEST-model algorithm A_2' that solves essentially the same task as the LOCAL-model algorithm A_2 from prior work, with the same asymptotic round complexity. We present the new CONGEST-model algorithm A_2' in Section 4 (more precisely, we develop a family of such algorithms, one for each gap in the complexity landscape).

Equivalence in trees: sublogarithmic region (Section 5). The preliminary observations in Section 3 cover the lowest parts of the complexity spectrum, and Section 4 covers the higher parts. To complete the proof of the equivalence of CONGEST and LOCAL in trees we still need to show the following result:

- Given: a randomized $o(\log n)$-round algorithm A_1 that solves some LCL Π in trees in the LOCAL model.
- We can construct: a randomized $O(\log \log n)$-round algorithm A_2' that solves it in the CONGEST model.

If we only needed to construct a LOCAL-model algorithm, we could directly apply the strategy from prior work [20, 23]: replace A_1 with a faster algorithm A_0 that has got a higher failure probability, use the Lovász local lemma (LLL) to show that A_0 nevertheless succeeds at least for some assignment of random bits, and then plug in an efficient distributed LLL algorithm [20] to find such an assignment of random bits.

However, there is one key component missing if we try to do the same in the CONGEST model: a sufficiently fast LLL algorithm. Hence we again arrive at a concrete algorithm design challenge: we need to develop an efficient CONGEST-model algorithm that solves (at least) the specific LLL instances that capture the task of finding a good random bit assignments for A_0.

We present our new algorithm in Section 5. We make use of the shattering framework of [28], but one of the key new ideas is that we can use the equivalence results for the superlogarithmic region that we already proved in Section 4 as a tool to design fast CONGEST model algorithms also in the sublogarithmic region.

Separation in general graphs (Section 6). Our third major contribution is the separation result between CONGEST and LOCAL for general graphs – and as we are interested in proving such a separation for LCLs, we need to prove the separation for bounded-degree graphs.

Our separation result is constructive – we show how to design an LCL problem Π with the following properties:

(a) There is a deterministic algorithm that solves Π in the LOCAL model in $O(\log n)$ rounds.
(b) Any algorithm (deterministic or randomized) that solves Π in the CONGEST model requires $\Omega(\sqrt{n}/\log^2 n)$ rounds.

To define Π, we first construct a graph family $\mathcal{G}$ and an LCL problem Π^{real} such that Π^{real} would satisfy properties (a) and (b) *if we promised that the input comes from family* $\mathcal{G}$. Here we use a bounded-degree version of the lower-bound construction by [38]: the graph has a small diameter, making all problems easy in LOCAL, but all short paths from one end to the other pass through the top of the structure, making it hard to pass a large amount of information across the graph in the CONGEST model.

However, the existence of LCL problems that have a specific complexity given some arbitrary promise is not yet interesting – in particular, LCLs with an arbitrary promise do not lead to any meaningful complexity classes or useful structural theorems. Hence the key challenge is *eliminating the promise* related to the structure of the input graph. To do that, we introduce the following LCL problems:

- Π^{proof} is a distributed proof for the fact $G \in \mathcal{G}$. That is, for every $G \in \mathcal{G}$, there exists a feasible solution X to Π^{proof}, and for every $G \notin \mathcal{G}$, there is no solution to Π^{proof}. This problem can be hard to solve.
- Π^{bad} is a distributed proof for the fact that a given labeling X is *not* a valid solution to Π^{proof}. This problem has to be sufficiently easy to solve in the LOCAL model whenever X is an invalid solution (and impossible to solve whenever X is a valid solution).

Finally, the LCL problem Π captures the following task:

- Given a graph G and a labeling X, solve either Π^{real} or Π^{bad}.

Now given an arbitrary graph G (that may or may not be from $\mathcal{G}$) and an arbitrary labeling X (that may or may not be a solution to Π^{proof}), we can solve Π efficiently in the LOCAL model as follows:

- If X is not a valid solution to Π^{proof}, we will detect it and we can solve Π^{bad}.
- Otherwise X proves that we must have $G \in \mathcal{G}$, and hence we can solve Π^{real}.

Particular care is needed to ensure to that even for an adversarial G and X, at least one of Π^{real} and Π^{bad} is always sufficiently easy to solve in the LOCAL model. A similar high-level strategy has been used in prior work to e.g. construct LCL problems with a particular complexity in the LOCAL model, but to our knowledge the specific constructions of $\mathcal{G}$, Π^{real}, Π^{proof}, and Π^{bad} are all new – we give the details of the construction in Section 6.

2 Preliminaries & Definitions

Formal LCL definition. An LCL problem Π is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C, r)$ satisfying the following.

- Both Σ_{in} and Σ_{out} are constant-size sets of labels;
- The parameter r is an arbitrary constant, called *checkability radius* of Π;
- C is a finite set of pairs $(H = (V^H, E^H), v)$, where:
  - H is a graph, v is a node of H, and the radius of v in H is at most r;
  - Every pair $(v, e) \in V^H \times E^H$ is labeled with a label in Σ_{in} and a label in Σ_{out}.

Solving a problem Π means that we are given a graph where every node-edge pair is labeled with a label in Σ_{in}, and we need to assign a label in Σ_{out} to each node-edge pair, such that every r-radius ball around each node is isomorphic to a (labeled) graph contained in C. We may use the term *half-edge* to refer to a node-edge pair.

LOCAL model. In the LOCAL model [31], we have a connected input graph G with n nodes, which communicate unbounded-size messages in synchronous rounds according to the links that connect them, initially known only to their endpoints. A trivial and well-known observation is that a T-round algorithm can be seen as a mapping from radius-T neighborhoods to local outputs.

CONGEST model. The CONGEST model is in all other aspects identical to the LOCAL model, but we limit the message size: in a network with n nodes, the size of each message is limited to at most $O(\log n)$ bits [36].

Randomized algorithms. We start by formally defining what is a randomized algorithm in our context. We consider randomized Monte Carlo algorithms, that is, the bound on their running time holds deterministically, but they are only required to produce a valid solution with high probability of success.

▶ **Definition 1** (Randomized Algorithm). *A randomized algorithm $\mathcal{A}$ run with parameter n, written as $\mathcal{A}(n)$, (known to all nodes) has runtime $t_{\mathcal{A}}(n)$ and is correct with probability $1/n$ on any graph with at most n nodes. There are no unique IDs. Further, we assume that there is a finite upper bound $h_{\mathcal{A}}(n) \in \mathbb{N}$ on the number of random bits that a node uses on a graph with at most n vertices.*

The local failure probability *of a randomized algorithm $\mathcal{A}$ at a node v when solving an* LCL *is the probability that the* LCL *constraint of v is violated when executing $\mathcal{A}$.*

The assumption that the number of random bits used by a node is bounded by some (arbitrarily fast growing) function $h(n)$ is made in other gap results in the LOCAL model as well (see e.g. "the necessity of graph shattering" in [22]). Our results do not care about the growth rate of $h(n)$, e.g., it could be doubly exponential in n or even growing faster. Its growth rate only increases the leading constant in our runtime.

The assumption that randomized algorithms are not provided with unique IDs is made to keep our proofs simpler, but it is not a restriction. In fact, any randomized algorithm can, in 0 rounds, generate an ID assignment, where IDs are unique with high probability. Hence, any algorithm that requires unique IDs can be converted into an algorithm that does not require them, by first generating them and then executing the original algorithm. The ID generation phase may fail, and the algorithm may not even be able to detect it and try to recover from it, but this failure probability can be made arbitrarily small, by making the ID space large enough. Hence, we observe the following.

▶ **Observation 2.** *Let $c \geq 1$ be a constant. For any randomized Monte Carlo algorithm with failure probability at most $1/n^c$ on any graph with at most n nodes that relies on IDs from an ID space of size n^{c+2} there is a randomized Monte Carlo algorithm with failure probability $2/n^c$ that does not use unique IDs.*

Deterministic algorithms. Differently from randomized algorithms, deterministic ones are not allowed to use randomness, and nodes are instead equipped with unique identifiers.

▶ **Definition 3** (Deterministic Algorithm). *A deterministic algorithm $\mathcal{A}$ run with parameter n, written as $\mathcal{A}(n)$, (known to all nodes) has runtime $t_{\mathcal{A}}(n)$ and is always correct on any graph with at most n nodes. We assume that vertices are equipped with unique IDs from a space of size $\mathcal{S}$. We require that a single ID can be sent in a* CONGEST *message. If the parameter $\mathcal{S}$ is omitted, then it is assumed to be n^c, for some constant $c \geq 1$.*

3 Warm-Up: The $O(\log n)$ Region

As a warm-up, we consider the regime of sublogarithmic complexities. In this region, we can use simple observations to show that gaps known for LCL complexities in the LOCAL model directly extend to the CONGEST model as well. Note that the results in this sections are immediate corollaries of previous work. In the following, we assume that the size of the ID space $\mathcal{S}$ is polynomial in n.

We start by noticing that a constant time (possibly randomized) algorithm for the LOCAL model implies a constant time deterministic algorithm for the CONGEST model as well.

▶ **Theorem 4** (follows from [23, 34])**.** *Let Π be an* LCL *problem. Assume that there is a randomized algorithm for the* LOCAL *model that solves Π in $O(1)$ rounds with failure probability at most $1/n$. Then, there is a deterministic algorithm for the* CONGEST *model that solves Π in $O(1)$ rounds.*

Proof. It is known that, the existence of a randomized $O(1)$-round algorithm solving Π in the LOCAL model with failure probability at most $1/n$ implies the existence of a deterministic algorithm solving Π in the LOCAL model in $O(1)$ rounds [23, 34].

Also, it is known that any algorithm $\mathcal{A}$ running in T rounds in the LOCAL model can be normalized, obtaining a new algorithm $\mathcal{A}'$ that works as follows: first gather the T-radius ball neighborhood, and then, without additional communication, produce an output. Since $\Delta = O(1)$, algorithm $\mathcal{A}'$ can be simulated in the CONGEST model. ◀

We can use Theorem 4 to show that, if we have algorithms that lie inside know complexity gaps of the LOCAL model, we can obtain fast algorithms that work in the CONGEST model as well.

▶ **Corollary 5.** *Let Π be an* LCL *problem. Assume that there is a randomized algorithm for the* LOCAL *model that solves Π in $o(\log\log^* n)$ rounds with failure probability at most $1/n$. Then, there is a deterministic algorithm for the* CONGEST *model that solves Π in $O(1)$ rounds.*

Proof. In the LOCAL model, it is known that an $o(\log\log^* n)$-round randomized algorithm implies an $O(1)$-round deterministic algorithm [23, 34]. Then, by applying Theorem 4 the claim follows. ◀

While Corollary 5 holds for any graph topology, in trees, paths and cycles we obtain a better result.

▶ **Corollary 6.** *Let Π be an* LCL *problem on trees, paths, or cycles. Assume that there is a randomized algorithm for the* LOCAL *model that solves Π in $o(\log^* n)$ rounds with failure probability at most $1/n$. Then, there is a deterministic algorithm for the* CONGEST *model that solves Π in $O(1)$ rounds.*

Proof. In the LOCAL model, it is known that, on trees, paths, or cycles, an $o(\log^* n)$-round randomized algorithm implies an $O(1)$-round deterministic algorithm [16, 34]. Then, by applying Theorem 4 the claim follows. ◀

We now show that similar results hold even in the case where the obtained algorithm does not run in constant time.

▶ **Theorem 7** (follows from [22, 32])**.** *Let Π be an* LCL *problem. Assume that there is a deterministic algorithm for the* LOCAL *model that solves Π in $o(\log n)$ rounds, or a randomized algorithm for the* LOCAL *model that solves Π in $o(\log\log n)$ rounds with failure probability at most $1/n$. Then, there is a deterministic algorithm for the* CONGEST *model that solves Π in $O(\log^* n)$ rounds.*

Proof. It is known that for solving LCLs in the LOCAL model, any deterministic $o(\log n)$-round algorithm or randomized $o(\log\log n)$-round algorithm can be converted into a deterministic $O(\log^* n)$-round algorithm [22]. We exploit the fact that the speedup result of [22] produces algorithms that are structured in a normal form. In particular, all problems solvable in $O(\log^* n)$ can also be solved as follows:

1. Find a distance-k $O(\Delta^{2k})$-coloring, for some constant k.
2. Run a deterministic k rounds algorithm.

The first step can be implemented in the CONGEST model by using e.g. Linial's coloring algorithm [32]. Then, similarly as discussed in the proof of Theorem 4, any T rounds algorithm can be normalized into an algorithm that first gathers a T-radius neighborhood and then produces an output without additional communication. Hence, also the second step can be implemented in the CONGEST model in $O(k) = O(1)$ rounds. ◀

We use this result to show stronger results for paths and cycles.

▶ **Corollary 8.** *Let Π be an* LCL *problem on paths or cycles. Assume that there is a randomized algorithm for the* LOCAL *model that solves Π in $o(n)$ rounds with failure probability at most $1/n$. Then, there is a deterministic algorithm for the* CONGEST *model that solves Π in $O(\log^* n)$ rounds.*

Proof. In the LOCAL model, it is known that, on paths and cycles, an $o(n)$-round randomized algorithm implies an $O(\log^* n)$-round deterministic algorithm [22]. The claim follows by applying Theorem 7. ◀

4 Trees: The $\Omega(\log n)$ Region

In this section we prove that in the regime of complexities that are at least logarithmic, the asymptotic complexity to solve any LCL on trees is the same in the LOCAL and in the CONGEST model, when expressed as a function of n. Combined with the results proved in Section 3 and Section 5, which hold in the sublogarithmic region, we obtain that the asymptotic complexity of any LCL on trees is identical in LOCAL and CONGEST.

Polynomial and subpolynomial gaps on trees. Informally, the following theorem states that there are no LCLs on trees with complexity between $\omega(\log n)$ and $n^{o(1)}$, and for any constant integer $k \geq 1$, between $\omega(n^{1/(k+1)})$ and $o(n^{1/k})$, in both the CONGEST and LOCAL models, and that the complexity of any problem is the same in both models.

▶ **Theorem 9** (superlogarithmic gaps)**.** *Let $T^{\text{slow}} = \{n^{o(1)}\} \cup \{o(n^{1/k}) \mid k \in \mathbb{N}^+\}$. For $k \geq 1$, let $f(o(n^{1/k})) := O(n^{1/(k+1)})$. Also, let $f(n^{o(1)}) := O(\log n)$.*

Let $T \in T^{\text{slow}}$. Let Π be any LCL *problem on trees that can be solved with a T-round randomized* LOCAL *algorithm that succeeds with probability at least $1 - 1/n$ on graphs of at most n nodes. The problem Π can be solved with a deterministic $f(T)$-round* CONGEST *algorithm.*

Given the description of Π it is decidable whether there is an $f(T)$-round deterministic CONGEST *algorithm, and if it is the case then it can be obtained from the description of Π.*

▶ Remark 10. The deterministic complexity in Theorem 9 suppresses an $O(\log^* |\mathcal{S}|)$ dependency on the size $|\mathcal{S}|$ of the ID space. Also, there is an absolute constant $l_\Pi = O(1)$ such that the CONGEST algorithm works even if, instead of unique IDs, a distance-l_Π input coloring from a color space of size $|\mathcal{S}|$ is provided.

Note that gap theorems do not hold if one has promises on the input of the LCL. For example, consider a path, where some nodes are marked and others are unmarked, and the problem requires to 2-color unmarked nodes. If we have the promise that there is at least one marked node every $\sqrt{n}$ steps, then we obtain a problem with complexity $\Theta(\sqrt{n})$, that does not exist on paths for LCLs without promises on inputs.

Since Theorem 9 shows how to construct CONGEST algorithms starting from LOCAL ones, the existence of CONGEST problems with complexity $\Theta(n^{1/k})$ follows implicitly from the existence of these complexities in the LOCAL model. In particular, Chang and Pettie devised a series of problems that they name $2\frac{1}{2}$-coloring [23]. These problems are parameterized by an integer constant $k > 1$ and have complexity $\Theta(n^{1/k})$ on trees.

Diameter time algorithms. Additionally, we prove that a randomized diameter time LOCAL algorithm is asymptotically not more powerful than a deterministic diameter time CONGEST algorithm, when solving LCLs on trees. This result can be seen as an orthogonal result to the remaining results that we prove for LCLs on trees, because the runtime is not expressed as a function of n, but as a function of a different parameter, that is, the diameter of the graph. While the result might be of independent interest, it mainly deals as a warm-up to explain the proof of the technically more involved Theorem 9.

▶ **Theorem 11** (diameter algorithms). *Let Π be an LCL problem on trees that can be solved with a randomized LOCAL algorithm running in $O(D)$ rounds that succeeds with high probability, where D is the diameter of the tree. The problem Π can be solved with a deterministic CONGEST algorithm running in $O(D)$ rounds. The CONGEST algorithm does not require unique IDs but a means to break symmetry between adjacent nodes, that can be given by unique IDs, an arbitrary input coloring, or an arbitrary orientation of the edges.*

Any solvable LCL problem on trees can trivially be solved in the LOCAL model in $O(D)$ rounds by gathering the whole tree topology at a leader node, who then locally computes a solution and distributes it to all nodes. Using pipelining, the same algorithm can be simulated in the CONGEST model in $O(D + n) = O(n)$ rounds, but this running time can still be much larger than the $O(D)$ running time obtainable in the LOCAL model. On a high level, we show that for LCLs on trees, it is not required to gather the whole topology at a single node and brute force a solution – gathering the whole information at a single node has an $\Omega(n)$ lower bound even if the diameter is small.

Black-white formalism. In order to keep our proofs simple, we consider a simplified variant of LCLs, called *LCLs in the black-white formalism.* The main purpose of this formalism is to reduce the radius required to verify if a solution is correct. We will later show that, on trees, the black-white formalism is in some sense equivalent to the standard LCL definition.

A problem Π is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, C_W, C_B)$ satisfying the following.

- Both Σ_{in} and Σ_{out} are constant-size sets of labels;
- C_B and C_W are sets of multisets of pairs of labels, where each pair (i, o) satisfies $i \in \Sigma_{\text{in}}$ and $o \in \Sigma_{\text{out}}$.

Solving a problem Π means that we are given a bipartite two-colored graph where every edge is labeled with a label in Σ_{in}, and we need to assign a label in Σ_{out} to each edge, such that for every black (resp. white) node, the multiset of pairs of input and output labels assigned to the incident edges is in C_B (resp. C_W).

Node-edge formalism. The black-white formalism allows us to define problems also on graphs that are not bipartite and two-colored, as follows. Given a graph G, we define a bipartite graph H, where white nodes correspond to nodes of G, and black nodes correspond to edges of G. A labeling of edges of H corresponds to a labeling of node-edge pairs of G. The constraints C_W of white nodes of H correspond to node constraints of G, and the constraints C_B of black nodes of H corresponds to edge constraints of G.

Equivalence on trees. Clearly, any problem that can be defined with the node-edge formalism can be also expressed as a standard LCL. Claim 12 states that the node-edge formalism and the standard LCL formalism are in some sense equivalent, if we restrict to trees.

▷ Claim 12. For any LCL problem Π with checkability radius r we can define an *equivalent* node-edge checkable problem Π'. In other words, given a solution for Π', we can find, in $O(r)$ rounds, a solution for Π, and vice versa.

By Claim 12, on trees, all LCLs can be converted into an equivalent node-edge checkable LCL, and note that the node-edge formalism is a special case of the black-white formalism where black nodes have degree 2. To make our proofs easier to read, in the rest of the section we prove our results in the black-white formalism (where the degree of black nodes is 2), but via Claim 12 all results also hold for the standard definition of LCLs. We start by proving Theorem 11.

Proof of Theorem 11. Recall that in the black and white formalism nodes are properly 2-colored, input and output labels are on edges (that is, there is only one label for each edge, and not one for each node-edge pair), and the correctness of a solution can be checked independently by black and white nodes by just inspecting the labeling of their incident edges. Assume we are given a tree. We apply the following algorithm, that is split into 3 phases.

1. *Rooting the tree.* By iteratively "removing" nodes with degree one from the tree, nodes can produce an orientation that roots the tree in $O(D)$ CONGEST rounds. This operation can be performed even if, instead of IDs, nodes are provided with an arbitrary edge orientation. In the same number of rounds, nodes can know their distance from the (computed) root in the tree. We say that nodes with the same distance to the root are in the same *layer*, where leaves are in layer 1, and the root is in layer $L = O(D)$.
2. *Propagate label-sets up.* We process nodes layer by layer, starting from layer 1, that is, from the leaves. Each leaf u of the tree tells its parent v which labels for the edge $\{u, v\}$ would make them happy. The set of these labels is what we call a *label-set.* A leaf u is *happy* with a label if the label satisfies u's LCL constraints in Π. Then, in round i, each node v in layer i receives from all children the sets of labels that make them happy, and tells to its parent w which labels for the edge $\{v, w\}$ would make it happy, that is, it also sends a *label-set.* Here v is *happy* with a label for the edge $\{v, w\}$ if it can label all the edges $\{\{v, u\} \mid u \text{ is a child of } v\}$ to its children such that all of its children are happy. In other words, it must hold that for any element in the label-set sent by v to w, there must exist a choice in the sets previously sent by the children of v to v, such that the constraints of Π are satisfied at v. In $O(D)$ rounds, this propagation of sets of label-sets reaches the root of the tree.
3. *Propagate final labels down.* We will later prove that, if Π is solvable, then the root can pick labels that satisfy its own LCL constraints and makes all of its children happy. Then, layer by layer, in $O(D)$ iterations, the vertices pick labels for all of their edges to their children such that their own LCL constraints are satisfied and all their children are happy. More formally, from phase 2 we know that, for any choice made by the parent, there always exists a choice in the label-sets previously sent by the children, such that the LCL constraints are satisfied on the node.

This algorithm can be implemented in $O(D)$ rounds in CONGEST as the label-sets that are propagated in the second phase are subsets of the constant size alphabet Σ. In the third phase, each vertex only has to send one final label per outgoing edge to its children.

Let $L_{v,u}$ be the label-set received by node v from its child u. We now prove that, if Π is solvable, then there is a choice of labels, in the label-sets received by the root, that makes the root happy. Since Π is solvable, then there exists an assignment ϕ of labels to the edges of the tree such that the constraints of Π are satisfied on all nodes. We prove, by induction on the layer number j, that every edge $\{u, v\}$ such that u is in layer j and v is the parent of u, satisfies that $\phi(\{u, v\}) \in L_{v,u}$. For $j = 1$ the claim trivially holds, since leaves send to their parent the set of all labels that make them happy. For $j > 1$, consider some node v in layer j. Let $u_1, \ldots, u_d$ be its children. By the induction hypothesis it holds that $\phi(\{v, u_i\}) \in L_{v,u_i}$. Let p be the parent of v (if it exists). Since ϕ is a valid labeling, and since it is true that if the parent labels the edge $\{v, p\}$ with label $\phi(\{v, p\})$ then v can pick something from its sets L_{v,u_i} and be happy, then the algorithm puts $\phi(\{v, p\})$ into the set $L_{p,v}$. Hence, every set L_{r,u_i} received by the root r contains the label $\phi(\{r, u_i\})$, implying that there is a valid choice for the root. ◀

While this process is extremely simple, its runtime of $O(D)$ rounds is rather slow. In order to obtain the $O(n^{1/k})$ and $O(\log n)$ CONGEST algorithms required for proving Theorem 9, we need a more sophisticated approach. As done by prior work in [21, 23], we decompose the tree into *fewer* layers, and show that, the mere existence of a fast algorithm for the problem implies that, similar to the algorithm of Theorem 11, it is sufficient to propagate constant sized label-sets between the layers; thus we obtain a complexity that only depends on the number of layers and their diameter. We proceed as follows.

5 Trees: $o(\log n)$ Randomized Implies $O(\log \log n)$

In this section we show that, on trees, any randomized algorithm solving an LCL problem Π in $o(\log n)$ rounds can be transformed into a randomized algorithm that solves Π in $O(\log \log n)$ rounds. This implies that in the CONGEST model there is no LCL problem in trees with a randomized complexity that lies between $\omega(\log \log n)$ and $o(\log n)$. Moreover, we show that it is not necessary to start from an algorithm for the CONGEST model, but that a LOCAL model one is sufficient. More formally, we will prove the following theorem.

▶ **Theorem 13** (sublogarithmic gap). *Let $c \geq 1$ be a constant. Given any* LCL *problem Π, if there exists a randomized algorithm for the* LOCAL *model that solves Π on trees in $o(\log n)$ rounds with failure probability at most $1/n$, then there exists a randomized algorithm for the* CONGEST *model that solves Π on trees in $O(\log \log n)$ rounds with failure probability at most $1/n^c$.*

In Section 5 we present the high level idea of the proof of Theorem 13. The proof itself is split into three subsections, for which a road map appears at the end of Section 5.

Proof Idea for Theorem 13

In the LOCAL model it is known that, on trees, there are no LCL problems with randomized complexity between $\omega(\log \log n)$ and $o(\log n)$ [20, 23]. At a high level, we follow a similar approach in our proof. However, while some parts of the proof directly work in the CONGEST model, there are some challenges that need to be tackled in order to obtain an algorithm that runs in $O(\log \log n)$ that is actually bandwidth efficient. We now provide the high level idea of our approach.

The standard approach: expressing the problem as an LLL instance. As in the LOCAL model case, the high level idea is to prove that if a randomized algorithm for an LCL problem Π runs in $o(\log n)$ rounds, then we can make it run faster at the cost of increasing its failure probability. In this way, we can obtain a constant time algorithm $\mathcal{A}_0$ at the cost of a very large failure probability. This partially gives what we want: we need a fast algorithm with small failure probability, and now we have a very fast algorithm with large failure probability. One way to fix the failure probability issue is to derandomize the algorithm $\mathcal{A}_0$, that is, to find a random bit assignment satisfying that if we run the algorithm with this specific assignment of random bits then the algorithm does not fail. Ironically, we use a randomized algorithm to find such a random bit assignment.

▶ **Lemma 14** (informal version). *For any problem* Π *solvable in* $o(\log n)$ *rounds with a randomized* LOCAL *algorithm having failure probability at most* $1/n$*, there exists a constant time* LOCAL *algorithm* $\mathcal{A}_0$ *that solves* Π *with constant local failure probability* p*.*

It turns out that the problem itself of finding a good assignment of random bits such that the constant time algorithm $\mathcal{A}_0$ does not fail can be formulated as a Lovász Local Lemma (LLL) instance. In an LLL instance there are random variables and a set of *bad events* that depend on these variables. The famous Lovász Local Lemma [27] states that if the probability of each bad event is upper bounded by p, each bad event only shares variables with d other events and the *LLL criterion* $epd < 1$ holds, then there exists an assignment to the variables that avoids all bad events (a more formal treatment of the Lovász Local Lemma is contained in the full version). In our setting, the random variables are given by the random bits used by the vertices and each vertex v has a bad event $\mathcal{E}_v$ that holds if the random bits are such that v's constraints in Π are violated if $\mathcal{A}_0$ is executed with these random bits. We show that a large polynomial LLL criterion – think of $p(ed)^{30} < 1$ – holds. Thus, the Lovász Local Lemma implies that there exist *good random bits* such the LCL constraints of Π are satisfied for all nodes when using these bits in $\mathcal{A}_0$. In the LOCAL model it is known how to solve an LLL problem with such a strong LLL criterion efficiently. We show that the same holds in the CONGEST model, i.e., $O(\log\log n)$ CONGEST rounds are sufficient to find a good assignment of random bits. We point out that we do not give a general LLL algorithm in the CONGEST model but an algorithm that is tailored for the specific instances that we obtain. The constant time algorithm $\mathcal{A}_0$ executed with these random bits does not fail at any node.

We summarize the high level approach as follows: Given an LCL problem Π defined on trees and an $o(\log n)$-rounds randomized algorithm $\mathcal{A}$ for Π, we obtain a constant time algorithm $\mathcal{A}_0$ for Π and a new problem Π' of finding good random bits for $\mathcal{A}_0$. Problem Π' is defined on the same graph as problem Π. The algorithm $\mathcal{A}_0$ and the problem Π' only depend on Π and $\mathcal{A}$. We show that Π' is also an LCL problem. In problem Π' each node of the tree needs to output a bit string such that if the constant time algorithm $\mathcal{A}_0$ is run with the computed random bits, the problem Π is solved. We will show that Π' can be solved in $O(\log\log n)$ rounds, and note that once Π' is solved, one can run $\mathcal{A}(n_0)$ for $t_0 = t_{\mathcal{A}_0} = O(1)$ rounds to solve Π.

▶ **Lemma 15** (informal version). *The problem* Π'*, that is, the problem of finding a good assignment of random bits that allows us to solve* Π *in constant time, can be solved in* $O(\log\log n)$ *rounds in the* CONGEST *model.*

Problem Π' is defined on the same tree as Π but problem Π' has checking radius $r + t_0$. Thus, the *dependency graph* of the LLL instance is a power graph of the tree, or in other words the LLL instance is *tree structured*. Hence, in the LOCAL model, Π' can be solved in $O(\log\log n)$ rounds by using a $O(\log\log n)$ randomized LOCAL algorithm for tree structured LLL instances [20]. We cannot do the same here, as it is not immediate whether this algorithm works in the CONGEST model (it is only clear that its shattering phase works in CONGEST).

The shattering framework. Our main contribution is showing how to solve Π' in $O(\log\log n)$ rounds in a bandwidth-efficient manner. To design an $O(\log\log n)$-round algorithm for Π', we apply the shattering framework for LLL of [28], that works as follows. After a precomputation phase of $O(\log^* n)$ rounds, the shattering process uses $\operatorname{poly}\Delta = O(1)$ rounds (the exponent depends on t_0 and the checking radius of the LCL) to determine the random bits of some of the nodes. The crucial property is that all nodes with unset random bits form *small connected components* of size $N = \operatorname{poly}(\Delta) \cdot \log n = O(\log n)$; in fact, even all nodes that are close to nodes with unset random bits form small connected components $C_1, \ldots, C_k$. Furthermore, each connected component can be solved (independently) with an LLL procedure as well, with a slightly tighter polynomial criterion (e.g., $p(ed)^{15} < 1$). Note that, in order to solve these smaller instances, we need to use a deterministic algorithm.This is because, if we try to recursively apply a randomized LLL algorithm on the smaller instances (e.g. by using the randomized LOCAL algorithm of [25]) we get that each component can be solved independently in $O(\log N)$ rounds, but with failure probability $1/\operatorname{poly} N \gg 1/n$.

Our main contribution: solving the small instances in a bandwidth efficient manner. Since it seems that we cannot directly use an LLL algorithm to solve the small remaining instances $C_1, \ldots, C_k$, we follow a different route. We devise a deterministic CONGEST algorithm that we can apply on each of the components in parallel: In Theorem 9 we prove that, on trees, any randomized algorithm running in $n^{o(1)}$ rounds (subpolynomial in the number of nodes) in the LOCAL model can be converted into a deterministic algorithm running in $O(\log n)$ in the CONGEST model. We use this result here, to show that, the mere existence of the randomized LOCAL algorithm of [25], that fails with probability at most $1/\operatorname{poly} N$ and runs in $O(\log N)$ rounds in the LOCAL model, which fits the runtime requirement of Theorem 9, implies that Π' can be solved in $O(\log N) = O(\log\log n)$ CONGEST rounds deterministically on the components induced by unset bits. To apply Theorem 9 that only holds for LCL problems, we express the problem of completing the partial random bit assignment as a problem Π'', that intuitively is almost the same problem as Π', but allows some nodes to already receive bit strings as their input. We show that Π'' is a proper LCL.

Formally, there are several technicalities that we need to take care of. In particular, we do not want to provide any promises on the inputs of Π'', as Theorem 9 does not hold for LCLs with promises on the input. For example, we cannot guarantee in the LCL definition that the provided input, that is, the partial assignment of random bits, can actually be completed into a full assignment that is good for solving Π. On the other hand, if we just defined Π'' as the problem of completing a partial given bit string assignment, it might be unsolvable for some given inputs, and this would imply that an $n^{o(1)}$ time algorithm for this problem cannot exist to begin with, thus there would be no way to use Theorem 9.

In order to solve this issue, we define Π'' such that it can be solved fast even if the input is not *nice* (for some technical definition of nice). In particular, we make sure, in the definition of Π'', that if the input is nice then the only way to solve Π'' is to actually complete the partial assignment, while if the input is not nice, and only in this case, nodes are allowed to output *wildcards* $\star$; the constraints of nodes that see wildcards in their checkability radius are automatically satisfied. This way the problem is always solvable. For an efficient algorithm, we make sure that nodes can verify in constant time if a given input assignment is nice or not. We also show that inputs produced for Π'' in the shattering framework are always nice. The definition of Π'' and the provided partial assignment allows us to split the instance of Π' into many independent instances of Π'' of size $N = O(\log n)$. By applying Theorem 9 we get that [25] implies the existence of a deterministic CONGEST algorithm $\mathcal{B}$ for Π'' with complexity $O(\log N) = O(\log\log n)$.

▶ **Lemma 16** (informal version)**.** *There is a deterministic* CONGEST *algorithm to solve* Π'' *on any tree with at most* N *nodes in* $O(\log N)$ *rounds, regardless of the predetermined input.*

We apply algorithm $\mathcal{B}$ on each of the components $C_1, \ldots, C_k$ in parallel, and the solution of Π'' on each small components together with the random bit strings from the shattering phase yield a solution for Π'. This can then be transformed into a solution for Π on the whole tree by running the constant time algorithm $\mathcal{A}_0$ with the computed random bits, which completes our task and proves Theorem 13.

6 Separation for General Graphs

In this section we define an LCL problem Π on general bounded-degree graphs, and show that, while Π can be solved deterministically in $O(\log n)$ rounds in the LOCAL model, any randomized CONGEST algorithm requires $\Omega(\sqrt{n}/\log^2 n)$ rounds. On a high level, the section is structured as follows. We start by formally defining a family $\mathcal{G}$ of graphs of interest, and then we present a set $\mathcal{C}^{\mathsf{proof}}$ of local constraints, satisfying that a graph G is in $\mathcal{G}$ if and only if it can be labeled with labels from a constant-size set, such that all nodes satisfy the constraints in $\mathcal{C}^{\mathsf{proof}}$. We then define our LCL Π in the following way:

- there is a problem Π^{real} such that, on any correctly labeled graph $G \in \mathcal{G}$, nodes must solve Π^{real};
- on any labeled graph $G \notin \mathcal{G}$, nodes can either output a locally checkable proof that shows that there exists a node in G that does not satisfy some constraint in $\mathcal{C}^{\mathsf{proof}}$ (by solving some LCL problem that we call Π^{bad}), or solve Π^{real} (if possible).

Finally, we show lower and upper bounds for Π in the CONGEST and LOCAL model respectively. The challenging part is to express all these requirements as a proper LCL, while preventing nodes from "cheating", that is, on all graphs $G \in \mathcal{G}$, it must not be possible for nodes to provide a locally checkable proof showing that G is not in $\mathcal{G}$, while for any graph $G \notin \mathcal{G}$ it should be possible to produce such a proof within the required running time.

Informally, the graphs contained in the family $\mathcal{G}$ of graphs look like the following: we start from a 2-dimensional grid; we build a binary tree-like structure on top of each column i, and let r_i be the root of the tree-like structure in top of column i; we use another grid to connect all left-most nodes of these trees; finally, we build on top of these r_i nodes another binary tree-like structure, where r_i nodes are its leaves. Note that the graphs in $\mathcal{G}$ are the bounded-degree variant of the lower bound family of graphs of Das Sarma et al. [38], where we also add some edges that are necessary in order to make the construction locally checkable.

More formally, we prove the following theorem.

▶ **Theorem 17.** *There exists an* LCL *problem* Π *that can be solved in* $O(\log n)$ *deterministic rounds in the* LOCAL *model, that requires* $\Omega(\sqrt{n}/\log^2 n)$ *rounds in the* CONGEST *model, even for randomized algorithms.*

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-linear lower bounds for distributed distance computations, even in sparse networks. In *Proc. 30th International Symposium on Distributed Computing (DISC 2016)*, volume 9888 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2016. doi:10.1007/978-3-662-53426-7_3.

2 Nir Bachrach, Keren Censor-Hillel, Michal Dory, Yuval Efron, Dean Leitersdorf, and Ami Paz. Hardness of distributed optimization. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 238–247, 2019. doi:10.1145/3293611.3331597.

3 Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. The distributed complexity of locally checkable problems on paths is decidable. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 262–271. ACM Press, 2019. doi:10.1145/3293611.3331606.

4 Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.17.

5 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. In *Proc. 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2019)*, pages 481–497. IEEE, 2019. doi:10.1109/FOCS.2019.00037.

6 Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. Improved distributed lower bounds for MIS and bounded (out-)degree dominating sets in trees. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*. ACM Press, 2021. doi:10.1145/3465084.3467901.

7 Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Distributed lower bounds for ruling sets. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 365–376. IEEE, 2020. doi:10.1109/FOCS46700.2020.00042.

8 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studený, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*. ACM Press, 2021. doi:10.1145/3465084.3467934.

9 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. *Distributed Computing*, 2020. doi:10.1007/s00446-020-00375-2.

10 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, pages 299–308. ACM Press, 2020. doi:10.1145/3382734.3405715.

11 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, pages 1307–1318. ACM Press, 2018. doi:10.1145/3188745.3188860.

12 Alkida Balliu, Juho Hirvonen, Dennis Olivetti, and Jukka Suomela. Hardness of minimal symmetry breaking in distributed computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 369–378. ACM Press, 2019. doi:10.1145/3293611.3331605.

13 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theor. Comput. Sci.*, 751:2–23, 2018. doi:10.1016/j.tcs.2016.07.005.

14 Sebastian Brandt. An automatic speedup theorem for distributed problems. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, pages 379–388. ACM, 2019. doi:10.1145/3293611.3331611.

15 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symposium on Theory of Computing (STOC 2016)*, pages 479–488. ACM Press, 2016. doi:10.1145/2897518.2897570.

16 Sebastian Brandt, Jan Grebík, Christoph Grunau, and Václav Rozhoň. The landscape of distributed complexities on trees, 2021. Unpublished manuscript.

17 Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. LCL problems on grids. In *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pages 101–110. ACM Press, 2017. doi:10.1145/3087801.3087833.

18 Keren Censor-Hillel and Michal Dory. Distributed spanner approximation. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 139–148. ACM, 2018. doi:10.1145/3212734.3212758.

19 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and near-quadratic lower bounds for the CONGEST model. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *LIPIcs*, pages 10:1–10:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.10.

20 Y.-J. Chang, Q. He, W. Li, S. Pettie, and J. Uitto. The complexity of distributed edge coloring with small palettes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2018. doi:10.1137/1.9781611975031.168.

21 Yi-Jun Chang. The complexity landscape of distributed locally checkable problems on trees. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.18.

22 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput.*, 48(1):122–143, 2019. doi:10.1137/17M1117537.

23 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

24 Yi-Jun Chang, Jan Studený, and Jukka Suomela. Distributed graph problems through an automata-theoretic lens. In *Proc. 28th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2021)*, LNCS. Springer, 2021. doi:10.1007/978-3-030-79527-6_3.

25 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the Lovász local lemma and graph coloring. *Distributed Comput.*, 30(4):261–280, 2017. doi:10.1007/s00446-016-0287-6.

26 Michael Elkin. An unconditional lower bound on the time-approximation trade-off for the distributed minimum spanning tree problem. *SIAM Journal on Computing*, 36(2):433–456, 2006. doi:10.1137/s0097539704441058.

27 P. Erdős and L. Lovász. Problems and results on 3-chromatic hypergraphs and some related questions, 1973.

28 M. Fischer and M. Ghaffari. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, volume 91 of *LIPIcs*, pages 18:1–18:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

29 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and impossibilities for distributed subgraph detection. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 153–162. ACM, 2018. doi:10.1145/3210377.3210401.

30 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2012)*, pages 1150–1162. SIAM, 2012. doi:10.1137/1.9781611973099.91.

31 Nathan Linial. Distributive graph algorithms – global solutions from local data. In *Proc. Symp. on Foundations of Computer Science (FOCS 1987)*, pages 331–335, 1987. doi:10.1109/SFCS.1987.20.

32 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

33 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 565–573, 2014. doi:10.1145/2591796.2591850.

34 Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

35 Dennis Olivetti. Round Eliminator: a tool for automatic speedup simulation, 2020. URL: https://github.com/olidennis/round-eliminator.

36 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

37 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed minimum-weight spanning tree construction. *SIAM Journal on Computing*, 30(5):1427–1442, 2000. doi:10.1137/s0097539700369740.

38 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012. doi:10.1137/11085178x.

Randomized Local Fast Rerouting for Datacenter Networks with Almost Optimal Congestion

Gregor Bankhamer
Department of Computer Sciences, Universität Salzburg, Austria

Robert Elsässer
Department of Computer Sciences, Universität Salzburg, Austria

Stefan Schmid
TU Berlin, Germany
Faculty of Computer Science, Universität Wien, Austria

Abstract

To ensure high availability, datacenter networks must rely on local fast rerouting mechanisms that allow routers to quickly react to link failures, in a fully decentralized manner. However, configuring these mechanisms to provide a high resilience against multiple failures while avoiding congestion along failover routes is algorithmically challenging, as the rerouting rules can only depend on local failure information and must be defined ahead of time. This paper presents a randomized local fast rerouting algorithm for Clos networks, the predominant datacenter topologies. Given a graph $G = (V, E)$ describing a Clos topology, our algorithm defines local routing rules for each node $v \in V$, which only depend on the packet's destination and are conditioned on the incident link failures. We prove that as long as number of failures at each node does not exceed a certain bound, our algorithm achieves an asymptotically minimal congestion up to polyloglog factors along failover paths. Our lower bounds are developed under some natural routing assumptions.

2012 ACM Subject Classification Theory of computation → Approximation algorithms analysis; Theory of computation → Distributed algorithms; Networks → Data path algorithms

Keywords and phrases local failover routing, congestion, randomized algorithms, datacenter networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.9

Related Version *Full Version*: https://arxiv.org/abs/2108.02136

Funding Research supported by the Vienna Science and Technology Fund (WWTF), project ´ICT19-045 (WHATIF), 2020-2024.

1 Introduction

Due to the popularity of data-centric applications and distributed machine learning, datacenter networks have become a critical infrastructure of the digital society. To meet the resulting stringent dependability requirements, datacenter networks implement fast failover mechanisms that enable routers to react to link failures quickly and to reroute flows in a decentralized manner, relying on *static* routing tables which include conditional *local* failover rules. Such local failover mechanisms in the data plane can react to failures orders of magnitudes faster than traditional global mechanisms in the control plane which, upon failure, may recompute routing tables by running the routing protocol again [6, 12, 18].

Configuring fast failover mechanisms however is challenging under multiple link failures, as the failover behavior needs to be pre-defined, before the actual failures are known. In particular, rerouting decisions can only rely on local information, without knowledge of possible further failures downstream. Without precautions, a local failover mechanism may hence entail congestion or even forwarding loops, already under a small number of link failures, while these issues could easily be avoided in a centralized setting.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 9; pp. 9:1–9:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

More formally, resilience is achieved in two stages. First, we are given a graph $G = (V, E)$ describing an undirected network (without failures). Our task is to compute local failover rules for each node $v \in V$ which define for each packet arriving at v to which incident link it should be forwarded, based on the packet's destination (known as destination-based routing); these rules can be conditioned on the status of the links incident to v. Second, when an adversary fails multiple links in the network, packets are forwarded according to our pre-defined conditional rules. Our objective is to define the static routing tables (i.e., the *rulesets*) in the first stage such that desirable properties are preserved in the second stage, in particular, connectivity and a minimal congestion.

This paper studies fast failover algorithms tailored towards Clos topologies, and more specifically to *(multirooted) fat-trees* [1, 17, 22], the predominant datacenter networks. In particular, we consider a scenario where $n-1$ sources inject one indefinite flow each to a single destination. This scenario has already been studied intensively in the literature [2, 4, 11, 21]: it models practically important operations such as in-cast [15, 24], and is also theoretically interesting as it describes a particularly challenging situation because it is focused on a single destination which can lead to bottlenecks.

The goal is to ensure that each flow reaches its destinations even in the presence of a large number of link failures while minimizing congestion: to provide a high availability, it is crucial to avoid a high load (and hence delays and packet loss) on the failover paths. The problem is related to classic load-balancing problems such as balls-into-bins, however, our setting introduces additional dependencies in that failover rules need to define valid paths.

1.1 Our Contribution

This paper studies the theoretical question of how to configure local fast failover rules in Clos topologies such that connectivity is preserved and load is minimized even under a large number of failures.

Results in a Nutshell. We first derive a lower bound showing that by failing $O(n/\log n)$ edges, the adversary can create a load of $\Omega(\log n/\log\log n)$ w.h.p. in arbitrary topologies with n nodes. As a next step, we give a routing protocol for complete bipartite graphs that incorporates local failover rules and, for up to $O(n/\log n)$ link failures, achieves an almost minimal congestion (i.e., up to $(\log\log n)^2$ factor). We then use the derived results to construct a failover ruleset for the Clos topology with $L + 1 = \Theta(1)$ levels and degree k (cf the definition in Section 4.1 and the example in Figure 1). It is resilient to $O(k/\log k)$ link failures, while keeping the total load below $O\left(k^{L-1}\log k \cdot \log\log k\right)$ w.h.p. For a certain class of routing protocols, that only forward over shortest paths (according to the local view of the nodes, cf Definition 22) and exhibit a property we call *fairly balanced*, this is again optimal up to $(\log\log n)^2$ factors. This class of protocols is natural and reminiscent of the widely-deployed shortest-path routing protocol ECMP (equal-cost multipath) [16, 22].

Techniques. In this work, we are interested in rulesets that include randomization, and are robust against an adversary, which knows the algorithm and the routing destination, but not the random choices leading to the specific failover routes.

In our lower bound analysis we need to cover a wide variety of failover protocols. While the deterministic case is well-understood [4], and failover protocols based on the uniform distribution are easy to handle, a mixture of the both is non-trivial to analyze. We opt for a carefully crafted case-distinction that captures failover paths, which might be predicted by the adversary with good probability. For this, we exploit the properties of the subgraphs induced by the edges, which have a certain (high) probability to be chosen as failover links. In all the other cases, we are able to use the (not necessarily uniform) random placement of the loads initiated by a subset of source nodes, and apply a balls-into-bins style argument to show that at least one node will receive high load.

To develop an efficient protocol for the Clos topology, we exploit the fact that it contains multiple bipartite sub-graphs. The main algorithm combines the advantages of deterministic protocols and forwarding loop-freeness, with the resilience of randomized approaches. Our approach builds upon the Interval protocol in [2], which is designed for the clique. However, the adaptation of this approach to the Clos topology comes with multiple challenges that need to be solved. The approach of [2] models the load that nodes receive with the help of trees that are tailored towards the clique, and this method can not be extended to more complex topologies. To overcome this problem, we use a Markov chain to describe such loads and develop a general Markov chain result that might be of independent interest (see Theorem 15). For Markov chains with state space $\mathbb{N}$ that drift towards 0 and that can be modelled with Poisson trials, it states a concentration inequality for the sum of the first $r = \Omega(1)$ elements. Additionally, the protocol in [2] relies on splitting the nodes into partitions of similar size. Contrary to the clique, where the assignment of nodes to partitions can be arbitrary, this is challenging in the case of the Clos topology. Furthermore, for our analysis in the Clos network we need to consider the flows arriving at a certain node from lower and upper levels concurrently. This leads to dependencies, which prevents us from using standard techniques such as Chernoff bounds and the method of bounded differences. To overcome this problem, we uncover the failover edges step by step, and utilize an inductive approach over the increasingly small subtrees around the destination, bounding the number of flows entering the corresponding subtree. For the details see Section 4.3.

1.2 Related Work

Motivated by measurement studies of network-layer failures in datacenters, showing that especially link failures are frequent and disruptive [14], the problem of designing resilient routing mechanisms has received much attention in the literature over the last years, see e.g., [4, 5, 7–10] or the recent survey by Chiesa et al. [6].

In this paper, we focus on the important model in which we do not allow for packet header rewriting or maintain state at the routers, which rules out approaches such as link reversal and others [13, 20]. A price of locality for static rerouting mechanisms has first been shown by Feigenbaum et al. [9] and Borokhovich et al. [4], who proved that it is not always possible to locally reroute packets to their destination even if the underlying network remains connected after the failures. These impossibility results have recently been generalized to entire graph families by Foerster et al. [10]. On the positive side, Chiesa et al. showed that highly resilient failover algorithms can be realized based on arc-disjoint arborescence covers [5, 7, 8], an approach which generalizes traditional solutions based on spanning trees [23]. However, these papers only focus on connectivity and do not consider congestion on the resulting failover paths. Furthermore, while arborescence-based approaches have the advantage that they are naturally loop-free, they result in long paths (and hence likely high load) and are complex to compute.

Only little is known about local fast failover mechanisms that account for load. Pignolet et al. in [21] showed that when only relying on deterministic destination-based failover rules, an adversary can always induce a maximum edge load of $\Omega(\varphi)$ by cleverly failing φ edges; when failover rules can also depend on the source address, an edge load of $\Omega(\sqrt{\varphi})$ can still be generated, when failing φ many edges. In [11], Foerster et al. build upon [4, 21], and leverage

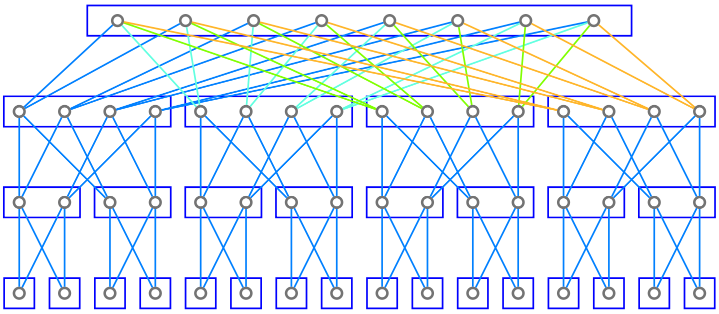


Figure 1 Clos topology with levels $0, 1, 2, 3$ (numbered from top to bottom) and degree $k = 4$.

a connection to distributed computing problems without communication [19], to devise a fast failover algorithm which balances load across arborescences using combinatorial designs. In these papers the focus is on deterministic algorithms.

Our work builds upon [2] where we showed that randomized algorithms can reduce congestion significantly in complete graphs. In particular, we presented three failover strategies: Assuming up to $\varphi = O(n)$ edge failures, the first algorithm ensures that w.h.p. a load of $O(\log n \log \log n)$ will not be exceeded at most nodes, while the remaining $O(\operatorname{polylog} n)$ nodes reach a load of at most $O(\operatorname{polylog} n)$. The second approach reduces the edge failure resilience to $O(n/\log n)$ but only requires knowledge of the packet destinations, and achieves a congestion of only $O(\log n \log \log n)$ at any node w.h.p. Finally, by assuming that the nodes do have access to $O(\log n)$ shared permutations of V, which are not known to the adversary, the node load can be reduced even further: a maximum load of only $O(\sqrt{\log n})$ occurs at *any* node w.h.p. However, our work relied on the assumption that the underlying network is fully connected (i.e., forms a clique). That said, simulations (performed after we published that paper) also showed promising first results for interval-based routing on Clos topologies.

In this work, we consider randomized fast failover specifically in the context of datacenter networks which typically rely on Clos topologies (also known as multi-rooted fat-trees) [1, 17, 22]. This scenario is not only of practical importance, but also significantly more challenging. Nevertheless, we are able to derive almost tight upper and lower bounds for this setting, under some natural fairness and shortest path assumptions.

1.3 Model

In a nutshell, our model includes two stages. First, we are asked to define the rulesets of the (static) routing tables of each node v in the network; these rules can depend on the destination and be conditioned on the possible link failures incident to v (i.e., only the *local* failures). Later, an adversary will decide which links to fail in the network; as the routing tables defined before are static and cannot be changed depending on the actual failures, packets will now simply be forwarded according to the local failover rules. Our objective is to pre-define these ruleset such that routing reachability is preserved under these failures and the load is minimized.

Local Destination-Based Failover Routing. We represent our network as an undirected graph $\mathcal{G} = (V, E)$ and denote by $\mathcal{F}$ the set of failed edges. Each node v is equipped with a static routing table α, which we assume to be precomputed without knowledge of $\mathcal{F}$. When a packet with destination d arrives at a node v, it is forwarded to the neighbor of v specified in the routing entry $\alpha(v, \mathcal{F}_v, d)$. Here $\mathcal{F}_v = \{w \mid (v, w) \in \mathcal{F}\}$ denotes the set of unreachable neighbors of v (which may be empty). In order to allow for randomization we assume that, for each node v, the entry $\alpha(v, \mathcal{F}_v, d)$ is drawn from $\mathcal{D}(v, \mathcal{F}_v, d)$, which is a distribution over $(\Gamma(v) \setminus \mathcal{F}_v) \cup \{v\}$. Here $\Gamma(v)$ denotes the set of neighbors of v. This way, a local failover routing protocol $\mathcal{P}$ can be described by the set of its distributions $\mathcal{P} = \{\mathcal{D}(v, \mathcal{F}_v, d) \mid v \in V, \mathcal{F}_v \subseteq \Gamma(v), d \in V\}$. We call such a protocol $\mathcal{P}$ *destination-based* as the only header information that is used for forwarding decisions is the destination address. In the following, we will also assume that for every v the edge (v, v) exists in $\mathcal{G}$ without being included in $\Gamma(v)$. This looping edge cannot be failed by the adversary and is used to enable the analysis of the case where $\mathcal{F}_v = \Gamma(v)$. Throughout the following sections, we assume the existence of an adversary, which knows the employed protocol, or in other words, the set of distributions $\mathcal{P}$ and may construct the set $\mathcal{F}$. However, this adversary does not know the random choices that lead to the routing entries α. In practice, to hide from the adversary longer term, this could be realized by generating new random tables once in a while.

Traffic Pattern and Load. Our focus lies on *flow*-based *all-to-one* routing [11,21]. That is, we assume that every node $v \in V \setminus \{d\}$ sends out an indefinite flow of packets towards some common destination d. For a node v, we will then say that it has *load* of ℓ (or short: $\mathcal{L}(v) = \ell$) iff ℓ such flows cross node v on their way to destination d. Similarly, for an edge e we define $\mathcal{L}(e)$ to denote the number of flows forwarded over edge e. In case some flow travels in a forwarding cycle, we say that all edges and nodes that lie on this cycle have infinite load.

Because we consider a purely destination-based ruleset, two flows hitting some node w will be forwarded via the same routing entry $\alpha(w, \mathcal{F}_w, d)$. Therefore, as soon as flows hit the same node they cannot be separated anymore. This implies that $\max_{v \in V \setminus \{d\}}\{\mathcal{L}(v)\} = \max_{e \in E}\{\mathcal{L}(e)\}$ as the node v with maximum load in $V \setminus \{d\}$ needs to forward all its flows over the edge (v, d) to reach the destination. As the edge load can be inferred from the node load, we will in the remaining part of our work only consider node loads.

1.4 Conventions and Structure

In the remainder of the paper, when we say we apply Chernoff bounds, we mean the usual multiplicative version. Furthermore, we denote by $\text{Bin}(n, p)$ the binomial distribution with n trials and success probability p, and by $\text{Unif}(A)$ the uniform distribution over elements in the set A. Finally, we denote by w.h.p. ("with high probability") a probability of at least $1 - n^{-\Omega(1)}$, where n is the networks size.

We start by presenting a lower bound in Section 2, stating that there exists a set of edge failures which induces a high load for any network and local destination-based failover protocol $\mathcal{P}$. In Section 3 we present an efficient loop-free failover protocol, which operates in complete bipartite graphs. This is used as a preliminary result to develop a protocol in Section 4, which may be employed in Clos topologies [1]. Each such section comes with a dedicated theorem and the corresponding analysis. Due to space constraints, we refer to the full version of our paper [3] for certain technical details.

2 Lower Bound for Local Destination-Based Failover Protocols

In the following section we construct a lower bound, stating that by failing $O(n/\log n)$ edges the adversary can w.h.p. always create a load of $\Omega(\log n/\log\log n)$.

▶ **Theorem 1.** *Consider any local destination-based failover protocol $\mathcal{P}$ that operates in a graph $\mathcal{G} = (V, E)$ with $|V| = n$ and assume that all nodes perform all-to-one routing to some node $d \in V$. Then, if d and $\mathcal{P}$ are known, a set of failures $\mathcal{F} \subseteq \{(v, d) \mid v \in V\}$ with $|\mathcal{F}| = O(n/\log n)$ can be constructed such that some node $v \neq d$ has $\mathcal{L}(v) > (1/10) \cdot \log n/\log\log n$ w.h.p.*

Most parts of our analysis are concerned with showing that the lower bound in Theorem 1 holds if $\mathcal{G}$ is the complete graph. Therefore, the following notation will be defined with this constraint in mind. We focus on an arbitrary but fixed destination-based failover protocol $\mathcal{P} = \{\mathcal{D}(v, \mathcal{F}_v, d) \mid v, d \in V, \mathcal{F}_v \subseteq V \setminus \{v\}\}$ and also fix the destination node $d \in V$. We will construct a set of failures $\mathcal{F}$ that induces a high load w.h.p. by only failing edges of the form (v, d), i.e., edges incident to the destination. This set of failures will have size $|\mathcal{F}| \leq \varepsilon \cdot n / \log n$, where $\varepsilon > 0$ is an arbitrary small constant. In this setting $\mathcal{F}_v$ – the set of unreachable neighbors of each node v – must either be $\{d\}$ or $\emptyset$ for any node v. We then abbreviate $\alpha(v, \{d\}, d)$ as $\alpha(v)$, which is the node to which v forwards its load in case the link (v, d) is failed. Similarly, we abbreviate the corresponding probability distribution $\mathcal{D}(v, \{d\}, d)$ as $\mathcal{D}_v$ and define f_v to denote the probability density function (PDF) of $\mathcal{D}_v$.

▶ **Definition 2** (Load Graphs). *The following directed graphs lie at the core of our analysis.*

1. $\mathcal{G}_\alpha^{\mathcal{F}} = (V \setminus \{d\}, E_\alpha^{\mathcal{F}})$ *where* $E_\alpha^{\mathcal{F}} = \{(v, \alpha(v, \mathcal{F}_v, d)) \mid v \in V \setminus \{d\}\}$
2. $\mathcal{G}_{\log} = (V \setminus \{d\}, E_{\log})$ *a directed graph with and* $E_{\log} := \{(v, w) \mid v \in V \setminus \{d\} \wedge f_v(w) > 1/\log^4 n\}$
3. $\mathcal{G}_{\log}^t$ *which is constructed from* $\mathcal{G}_{\log}$ *by removing edges in the following way:*
   a. *First, remove arbitrary (outgoing) edges from nodes v with* out-deg$(v) > 1$ *until every node has degree* ≤ 1.
   b. *Second, break any remaining cycle by removing an arbitrary edge from each cycle.*

The graph $\mathcal{G}_\alpha^{\mathcal{F}}$, given a set of failures $\mathcal{F}$ and destination d, describes the path that flows take. In order to fulfill our goal of creating a high load at some node v, we will make sure that some node v is reached by many nodes in $\mathcal{G}_\alpha^{\mathcal{F}}$. Note that this graph is a random variable as the entries in α follow distributions.

▶ **Observation 3.** *$\mathcal{L}(v) = \infty$ iff v lies on a cycle in $\mathcal{G}_\alpha^{\mathcal{F}}$. Otherwise $\mathcal{L}(v)$ is equal to the number of nodes $w \in V$ such that a path from w to v exists in $\mathcal{G}_\alpha^{\mathcal{F}}$.*

The graph $\mathcal{G}_{\log}$ allows us to capture whether the protocol $\mathcal{P}$ contains many failover edges that may be predicted by the adversary. Note, if the edge $(v, w) \in E_{\log}$ and (v, d) are failed, then v forwards its flows to w with probability larger than $1/\log^4 n$. Finally, $\mathcal{G}_{\log}^t$ is just a subgraph of $\mathcal{G}_{\log}$, which does not contain any cycles and simplifies our analysis in some cases. These graphs are related to $\mathcal{G}_\alpha^{\mathcal{F}}$ in the following way.

▶ **Observation 4.** *If $\mathcal{F}_v = \{d\}$, then $(v, w) \in \mathcal{G}_\alpha^{\mathcal{F}}$ with probability $f_v(w)$. This probability is independent of other edges (s, t) with $s \neq v$ being in $\mathcal{G}_\alpha^{\mathcal{F}}$. In case $(v, w) \in \mathcal{G}_{\log}$ (or $(v, w) \in \mathcal{G}_{\log}^t$) it follows that $f_v(w) > 1/\log^4 n$.*

Intuitively, if $\mathcal{G}_{\log}$ contains many edges, then we are in a setting close to determinstic failover protocols. By carefully failing edges of the form (v, d), we have a good chance to make them appear in $\mathcal{G}_\alpha^{\mathcal{F}}$ and create a node v which is reached by many other nodes. One final definition involves the natural definition of a reverse tree: it is reversed in the sense that all edges are oriented towards the root.

▶ **Definition 5** (Reverse Tree). *We call a directed graph $\mathcal{G}$* reverse tree *iff*

1. *there is a node r in $\mathcal{G}$ with* out-deg$(r) = 0$ *that can be reached from all nodes in $\mathcal{G}$, and*
2. *every node $v \neq r$ in $\mathcal{G}$ has* out-deg$(v) = 1$.

We call r the reverse root *of $\mathcal{G}$. Furthermore, we call a graph $\mathcal{G}'$* reverse subtree *of $\mathcal{G}$ iff $\mathcal{G}'$ is both, a reverse tree and a subgraph of $\mathcal{G}$.*

Note, from the construction of $\mathcal{G}^t_{\log}$ it follows that it is a reverse forest. Let the sets V_R and V^t_R contain the nodes of out-degree 0 in $\mathcal{G}_{\log}$ and $\mathcal{G}^t_{\log}$, respectively. Observation 4 implies for nodes in V_R that we cannot easily predict their forwarding targets. Additionally, when constructing $\mathcal{G}^t_{\log}$ (see Definition 2) in the first step, not a single node has its out-degree modified to 0. This can only happen in the second step, where exactly one node turns into a reverse root. Therefore, the difference $|V^t_R| - |V_R|$ is equal to the number of cycles that were removed in this second step.

Analysis Outline

In the following subsections of the analysis we focus on the complete graph. Depending on the structure of $\mathcal{G}_{\log}$ and $\mathcal{G}^t_{\log}$ we split our analysis into 3 cases. As this graphs are inferred from $\mathcal{P}$, this can also be seen as a distinction between different types of routing protocols. In Section 2.1, we consider the case $|V^t_R| - |V_R| \geq \sqrt{n}$, which intuitively corresponds to the case where $\mathcal{P}$ is prone to produce forwarding loops. In the second case (Section 2.2), we consider $|V_R| \geq \varepsilon n/\log n$, which implies that there are many nodes of degree 0 in $\mathcal{G}_{\log}$. Such nodes do not have a preferred forwarding target in case their link to d is failed. They behave similarly to nodes that forward their flows to neighbors selected uniformly at random. In the last case (Section 2.3), we consider $|V_R| \leq \varepsilon n/\log n$. In this case most nodes have at least one out-going edge in $\mathcal{G}_{\log}$, which can be exploited by the adversary. In any of the three cases, we show that, by failing at most $O(n/\log n)$ edges, a load of $(1/10) \cdot \log n/\log\log n$ is accumulated at some node in the network w.h.p. Finally, we give the proof of Theorem 1 in Section 2.4.

2.1 Analysis Case 1: $|V^t_R| - |V_R| \geq \sqrt{n}$

Recall, the condition of this case implies that $\mathcal{G}_{\log}$ contains at least n^{ε} many cycles. The idea is, to fail the edge (v, d) for many nodes v that lie on such a cycle. Then, either the whole cycle or at least a long path of nodes lying on such a cycle appears in $\mathcal{G}^{\mathcal{F}}_{\alpha}$ w.h.p. and causes high load. The detailed proof is given in the full version [3].

▶ **Lemma 6.** *There exists a set of failures $\mathcal{F} \subseteq \{(v, d) \mid v \text{ lies on a cycle in } \mathcal{G}_{\log}\}$ with $|\mathcal{F}| = \sqrt{n} \cdot (1/10) \cdot \log n/\log\log n$ such that some node v that lies on a cycle has $\mathcal{L}(v) \geq (1/10) \cdot \log n/\log\log n$ w.h.p.*

2.2 Analysis Case 2: $|V_R| \geq \varepsilon n/\log n$

In this setting, many nodes v have out-degree 0 in $\mathcal{G}_{\log}$. In case the link (v, d) of such a node is failed, it is hard to predict the failover edge $(v, \alpha(v))$ as these nodes have multiple potential forwarding targets. However, this can be exploited as there must be a set of nodes which are potential forwarding targets of many nodes in V_R. Similarly as in the analysis of a balls-into-bins process, we deduce that, w.h.p., there is one such node that receives load from $\Omega(\log n \cdot \log\log n)$ nodes in V_R. To simplify our analysis, we let V'_R be an arbitrary but fixed subset of V_R with size exactly $\varepsilon n/\log n$. A proof for the following statement is given in the full version [3]. Note that, if the second statement of the following lemma holds, we are already done.

▶ **Lemma 7.** *For $\mathcal{F} = \{(v, d) \mid v \in V'_R\}$ one of the following statements holds:*

1. *In expectation, at least $n^{1/8}$ many nodes $w \in V$ have each at least $(1/10) \cdot \log n/\log\log n$ incident edges that originate from V'_R.*
2. *W.h.p., there exists a node w with $\mathcal{L}(w) = \log^2 n(1 - o(1))$.*

The following statement can be shown with the help of standard-techniques, namely the *method of bounded differences* (sometimes also called McDiarmid inequality). This concentration inequality and a detailed proof is given in the full Version [3].

▶ **Lemma 8.** *Let $\mathcal{F} = \{(v,d) \mid v \in V_R'\}$ and assume that the first statement of Lemma 7 holds. Then, w.h.p., there exists a node v such that $\mathcal{L}(v) > (1/10) \cdot \log n / \log\log n$.*

2.3 Analysis Case 3: $|V_R| < \varepsilon n / \log n$

We assume throughout this section that the condition $|V_R^t| - |V_R| \geq \sqrt{n}$ does not hold as that case was already analysed in Section 2.1. This, however, implies that $|V_R^t| < |V_R| + \sqrt{n} < 2\varepsilon n/\log n$. Recall Definition 5 and that $|V_R^t|$ is the number of reverse roots; or in other words, the number of reverse trees in the forest $\mathcal{G}_{\log}^t$. The idea behind this section is simple. If, for the nodes of some reverse subtree in $\mathcal{G}_{\log}^t$ of size $(1/10) \cdot \log n \cdot \log\log n$, we fail the edges incident to destination d, then this whole subtree will appear in $\mathcal{G}_\alpha^{\mathcal{F}}$ with probability $\log^{-(4/10)\log n/\log\log n} n = n^{-4/10}$. By Observation 3 the root of this tree will then receive $(1/10) \cdot \log n / \log\log n$ load. The main challenge is to construct a large enough set of independent trees such that at least one of them appears in $\mathcal{G}_\alpha^{\mathcal{F}}$ w.h.p. This is where the following counting argument comes into play.

▶ **Observation 9.** *If $|V_R| < \varepsilon n/\log n$ and $|V_R^t| - |V_R| < \sqrt{n}$ then $\mathcal{G}_{\log}^t$ must contain one of the following*

1. *$\sqrt{n}$ disjoint reverse trees of size $> (1/10) \cdot \log n / \log\log n$ each, or*
2. *one reverse tree of size $> \sqrt{n}/2$.*

Proof. The proof follows by a counting argument. Assume both statements do not hold. Then, the number of nodes $\mathcal{G}_{\log}^t$ contains can be upper-bounded by

$$\left(\frac{2\varepsilon n}{\log n} - \sqrt{n}\right) \cdot (1/10) \frac{\log n}{\log\log n} + \sqrt{n} \cdot \frac{\sqrt{n}}{2} < n - 1.$$

The first product reflects that all but $\sqrt{n}$ trees have size at most $(1/10) \cdot \log n / \log\log n$. The second product reflects the worst-case of each of these at most $\sqrt{n}$ remaining trees having size $\sqrt{n}/2$. The above inequality chain leads to a contradiction as $\mathcal{G}_{\log}^t$ contains $n-1$ nodes. ◀

We now present a lemma for both of the cases in Observation 9, each achieving the lower bound in Theorem 1. The proofs follow the ideas sketched at the start of this section. In case of Lemma 11, the tree of size $\geq \sqrt{n}/2$ needs to be split into $\sqrt{n}/\operatorname{polylog} n$ node-disjoint subtrees of size $(1/10) \cdot \log n / \log\log n$. The proofs are given in the full version [3].

▶ **Lemma 10.** *Assume there are $\sqrt{n}$ reverse trees in $\mathcal{G}_{\log}^t$ of size at least $(1/10) \cdot \log n / \log\log n$ each. Then, there exists a failure set $\mathcal{F} \subseteq \{(v,d) \mid v$ lies on a tree in $\mathcal{G}_{\log}^t\}$ with $|\mathcal{F}| = \sqrt{n} \cdot (1/10) \cdot \log n / \log\log n$ such that a node v has $\mathcal{L}(v) > (1/10) \cdot \log n / \log\log n$.*

▶ **Lemma 11.** *Assume there is a reverse tree $\mathcal{T}_R$ of size $\sqrt{n}/2$ in $\mathcal{G}_{\log}^t$. Then, there exists a set of failures $\mathcal{F} \subseteq \{(v,d) \mid v$ lies in $\mathcal{T}_R\}$ with $|\mathcal{F}| \leq \sqrt{n} \cdot (1/10) \cdot \log n / \log\log n$ such that a node v of $\mathcal{T}_R$ has $\mathcal{L}(v) > (1/10) \cdot \log n / \log\log n$ w.h.p.*

2.4 Proof of Theorem 1

In Sections 2.1–2.3 we considered the complete graph $\mathcal{G}$ together with a fixed destination-based protocol $\mathcal{P}$ and all-to-one destination d. In this setting, we constructed the graphs $\mathcal{G}_{\log}, \mathcal{G}^t_{\log}$ and split our analysis into three cases, depending on the structure of these graphs. In each case, we establish that the theorem's result w.r.t. $\mathcal{G}$ holds:

1. Case 1: If the case in Section 2.1 occurs, then the result immediately follows from Lemma 6.
2. Case 2: If we are in the case of Section 2.2, then either the second statement of Lemma 7 holds and the result follows, otherwise the first statement holds and Lemma 8 leads to the desired result.
3. Case 3: This case was covered in Section 2.3, and further splits into two sub-cases as indicated in Observation 9. In both sub-cases, the result follows as stated in Lemmas 10 and 11, respectively.

The proof for general undirected graphs $\mathcal{G} = (V, E)$ with $|V| = n$ follows from Lemma 26, which is stated in the full version of our paper [3]. The basic idea is that, for any protocol $\mathcal{P}$ operating in $\mathcal{G}$, one can construct an equivalent protocol $\mathcal{P}_K$ that operates in the clique K_n (equivalent in the sense that the path flows take is the same in both graphs). We then use the statement of Theorem 1, which we already established for complete graphs, to deduce that a set of failures $\mathcal{F}^{(K)}$ exists that induces a high load in K_n. The same set of failures (excluding some edges which may not exist in $\mathcal{G}$) also leads to a high load in $\mathcal{G}$ when employing $\mathcal{P}$.

3 Interval Routing in the Bipartite Graph

In the following section, we will construct an efficient local failover protocol for the complete bipartite graph $G = (V \cup W, E)$. Here the set of nodes $V \cup W$ consists of two sets, where $|V| = |W| = n$ and edges are drawn such that each node $v \in V$ is connected to every $w \in W$ and vice versa.

To employ our routing protocol, we further assume that the nodes in both, V and W, are partitioned into $K := C \log n$ sets, where $C = \Theta(1)$ is an arbitrary value larger 4. That is, $V = V(0) \cup V(2) \cup ... \cup V(K-1)$ and $W = W(0) \cup W(2) ... \cup W(K-1)$, where we assume that all these partitions have size $I := n/K = n/(C \log n)$ (assume $C \log n$ divides n). We propose the following local routing protocol, which is resilient to $\Omega(n/\log n)$ edge failures.

▶ **Definition 12** (Bipartite Interval Routing). *We define the routing protocol $\mathcal{P}_B$, induced by the following distributions when routing towards some node $d \in W$*

- *For $v \in V(i)$ we set $\mathcal{D}(v, \mathcal{F}_v, d) = \mathrm{Unif}(\{d\})$ if $d \notin \mathcal{F}_v$, otherwise $\mathcal{D}(v, \mathcal{F}_v, d) = \mathrm{Unif}(W(i) \setminus \mathcal{F}_v)$.*
- *For $w \in W(i)$ with $w \neq d$ we set $\mathcal{D}(w, F_w, d) = \mathrm{Unif}(V((i+1) \mod K) \setminus \mathcal{F}_w)$.*

Note that this protocol is inspired by the *Interval* routing protocol of [2] which is constrained to complete graphs. Intuitively, a packet with source in the set $V(i)$ follows the partitions $V(i) \to W(i) \to V(i+1) \to W(i+1)...$ until reaching a node $v \in V$ such that (v, d) is not failed. Therefore, the only way for flows to end up in a cycle is by travelling through all $2K$ intervals, which is very unlikely. We may also refer to this alternation between layers V and W of a packet as "ping-pong" in the remainder of the paper.

▶ **Theorem 13.** *Let $G = (V \cup W, E)$ be a complete bipartite graph with $|V| = |W| = n$. Let the routing protocol $\mathcal{P}_B$ be employed, configured with $C > 4$, and all-to-one routing towards some destination $d \in W$ be performed. Assume the set of failures $\mathcal{F}$ fulfills for every i with $0 \leq i < K$ that*

1. *$\forall w \in W : |\{v \in V(i) \mid w \in \mathcal{F}_v\}| \leq I/3$, and*
2. *$\forall v \in V : |\{w \in W(i) \mid v \in \mathcal{F}_w\}| \leq I/3$.*

Then, with probability at least $1 - 3n^{-(C-1)}$, every node $u \in V \cup W$ with $u \neq d$ has $\mathcal{L}(u) = O(\log n \cdot \log\log n)$, even if $\mathcal{F}$ is constructed with knowledge of $\mathcal{P}_B$ and d.

Intuitively, the constraint on $\mathcal{F}$ states that at most a $(1/3)$ fraction of nodes in the same interval may have failed edges incident to the same node. The constraint is, for example, easily fulfilled in case only $I/3 = n/(3C \log n) = \Theta(n/\log n)$ edges are failed in total. This implies that the load induced by the protocol approaches the lower bound in Theorem 1 up to only a polyloglog n factor. In any deterministic protocol, a load of $\Omega(n/\log n)$ could be created in this setting [4]. Additionally, the simple randomized protocol, which forwards the packets between nodes of V and W which are selected uniformly at random until a node $v \in V$ is reached such that (v, d) is not failed, is prone to cycles. By failing $O(n/\log n)$ arbitrary edges between nodes in V and d, at least one flow will travel from such a node $v \in V$ to some $w \in W$ and back to v with probability $\geq 1/\operatorname{polylog} n$. This creates a forwarding loop of length 2 and prevents some flows from reaching destination d. Our interval protocol is hybrid in the sense that nodes forward their packet uniformly at random according to pre-determined partitions. This allows it to keep the network load low while also avoiding forwarding loops w.h.p.

3.1 Analysis of the Bipartite Interval Protocol

We consider a fixed destination node d together with a set of failures $\mathcal{F}$ that fulfills the requirements of Theorem 13. To make our analysis more readable, we assume that all partitions $V(i)$ and $W(j)$ have exactly the same size. Furthermore, we denote by $\alpha(v)$ the (random) node that v forwards packets towards destination d when following $\mathcal{P}_B$. Before starting with the proof of the theorem, we show the following important statement, which implies that, w.h.p., no flows travel in a cycle until they reach the destination d. Packets "ping-pong" between nodes in V and W until they reach the destination. Due to the restrictions on the failure set in Theorem 13 it follows that each time a packet lands on some node $v \in V$, there is a constant probability that the link (v, d) is not failed. It follows that, w.h.p., the packet reaches d after $K = \Theta(\log n)$ alternations between V and W. A detailed proof is given in the full version of our paper [3].

▶ **Lemma 14.** *Any packet starting at some node $u \in V \cup W$ will reach destination d in less than $2K$ hops with probability at least $1 - n^{-C}$.*

The other important ingredient in the proof of Theorem 13 is the following technical statement about Markov chains. A proof for this statement is given in the full version [3]. It exploits that, in expectation, the chain drifts towards 0 with every two further elements.

▶ **Theorem 15** (Markov Chain Aggregation). *Let $\{X_i\}_{i \geq 0}$ be a Markov chain over state space $\mathbb{N}_0$ and $\phi, \psi > 0$ be constants with $\phi \cdot \psi < 1$. Let the following be fulfilled for every $i > 0$:*

1. *X_i can be modeled by a sum of Poisson trials that only depends on X_{i-1}*
2. *$\mathbb{E}[X_{2i+1}] \leq X_{2i} \cdot \phi$*
3. *$\mathbb{E}[X_{2i}] \leq X_{2i-1} \cdot \psi$*

Then, there exists a constant $C_{\phi\psi} > 1$, such that for any fixed $r > C_{\phi\psi}$ it holds that $\sum_{i=0}^{r} X_i = O(\log(r) \cdot r)$ with probability at least $1 - 2\exp(-3r)$ as long as $X_0 = O(r)$.

Proof of Theorem 13. We let $V_G := \{v \mid v \in V \wedge (v,d) \notin \mathcal{F}\}$ denote the set of *good* nodes in V that may forward incoming packets directly to destination d. Note that each flow that eventually reaches d, does so over some $v \in V_G$. Therefore, in case no flow traverses a cycle, it follows that the node with maximum load will be some $v \in V_G$. In the following we will consider one such fixed node $r \in V_G$. W.l.o.g. we assume that this node lies in $V(K-1)$ such that we can avoid modulo operations. We define L_j to be the set of nodes whose load r receives within exactly j hops. Clearly $L_0 = \{r\}$, and according to the definition of $\mathcal{P}_B$ it must hold that $L_1 \subseteq W(K-2)$, $L_2 \subseteq V(K-2)$, $L_3 \subseteq W(K-3)$ and so forth. Our goal is to bound the values $|L_j|$ which allows us to determine the load that r receives. To that end, we initially assume that the routing entries $\alpha(v)$ of any node v have not yet been uncovered. Observe that, in order to determine, for example, L_1 it suffices to uncover the entries of nodes in $W(K-2)$ and check which nodes $w \in W(K-2)$ have $\alpha(w) = r$. To determine L_2, we then uncover entries in $V(K-2)$ and check the number of nodes v in this partition having $\alpha(v) \in L_1$. A repetition of this approach step-by-step yields the following intermediate result, which we show in the full version [3] in detail.

▶ **Observation 16.** *The sequence $\{|L_i|\}_{i=0}^{2K}$ forms a Markov chain with $|L_0| = 1$. Additionally, for $i > 0$ it holds that*

1. *$|L_i|$ can be modeled by a sum of Poisson trials depending only on $|L_{i-1}|$*
2. $\mathbb{E}\left[|L_{2i+1}|\right] \leq (3/2) \cdot |L_{2i}|$
3. $\mathbb{E}\left[|L_{2i}|\right] \leq (1/2) \cdot |L_{2i-1}|$

Next we make use of Lemma 14. Its statement, together with a union bound application, implies that *no* packet originating from any node travels more than $2K$ hops with probability at least $\geq 1 - |V \cup W| n^{-C} \geq 1 - 2n^{-(C-1)}$. This implies $|L_i| = 0$ for $i \geq 2K$. Hence, our fixed node $r \in V_G$ receives in total $\sum_{i=0}^{2K} |L_i|$ load w.h.p. As the sequence $\{|L_i|\}_{i\geq 0}^{2K}$ is a martingale that follows the properties described in Observation 16, we may apply the Markov chain result Theorem 15 for $r = 2K$. It implies that $\sum_{i=0}^{2K} |L_i| = O(\log n \cdot \log\log n)$ with probability at least $1 - 2n^{-6C}$. Hence, a union bound application yields that for *any* node $r \in V_G$, we have with probability at least $1 - n \cdot (2n^{-(C-1)} - 2n^{-6C}) > 1 - 3n^{-(C-1)}$ that $\mathcal{L}(r) = O(\log n \cdot \log\log n)$. As *no* packet starting at any node $V \cup W$ travels in a cycle, the node with maximum load (excluding d) must be some node $r \in V_G$ and Theorem 13 follows. ◀

3.2 Lower Bound for the Bipartite Graph

In the following, we present a different lower bound variant. It also holds in settings where nodes in W do not contribute one initial flow in the all-to-one routing process. However, it only guarantees high load in expectation as opposed to the high probability guarantee of Theorem 1. We will make use of this version in the analysis of the Clos topology. The proof is given in the full version [3].

▶ **Lemma 17.** *Let $G = (V \cup W, E)$ be a complete bipartite graph with $|V| = |W| = n$ and assume that the nodes in V each initiate one flow towards some node $d \in W$. Then, for any local destination-based failover protocol $\mathcal{P}$, there exists a set of failures $\mathcal{F}$ of size $|\mathcal{F}| \leq \varepsilon \cdot n/\log n$, $\varepsilon > 0$ arbitrary constant, such that, in expectation, the number of nodes with load $\Omega(\log n/\log\log n)$ is at least one.*

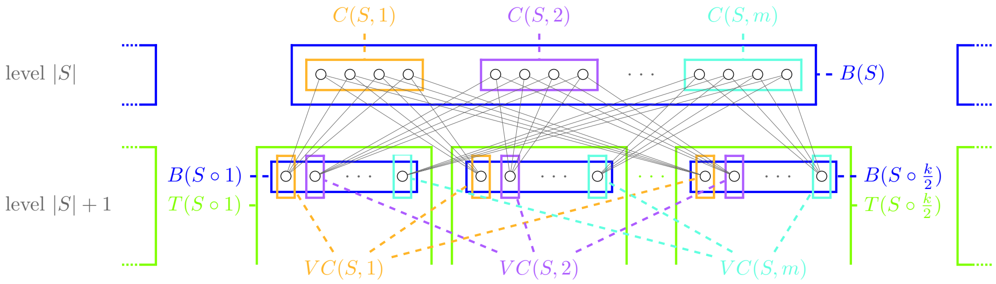


Figure 2 Links between a fixed block $\mathrm{B}\,(S)$ and its children. Here m denotes the number of clusters in block $\mathrm{B}\,(S)$.

4 Efficient Protocol for the Clos Topology

4.1 Topology Description

The Clos topology we consider comes with two parameters k, the degree of each node in the network, and $L+1$, $L \geq 1$, the number of levels in the network (cf. also Figure 1). It is constructed as follows. On level 0, there are $(k/2)^L$ many nodes and each level ℓ, $1 \leq \ell \leq L$, consists of $2(k/2)^L$ many nodes. We assume the nodes in each level to be numbered, starting with 1. All nodes are then partitioned into *blocks*. We denote such a block by $\mathrm{B}\,(S)$, where S is a sequence from the set $\mathbb{S}_L$. This set contains all sequences $S = (s_1, s_2, ..., s_\ell)$ of length $0 \leq \ell \leq L$, where the s_i are integers subject to $s_1 \in [1, k]$ and $s_i \in [1, k/2]$, $i > 1$. The nodes in level ℓ are contained in blocks $\mathrm{B}\,(S)$ with $S \in \mathbb{S}_L \wedge |S| = \ell$. Each such block contains $(k/2)^{L-\ell}$ many consecutive nodes of level ℓ. In level 0 there is only a single block. In case $\ell > 1$ and $S = (s_1, s_2, ..., s_\ell)$ the block $\mathrm{B}\,(S)$ contains the nodes $[o+1, o+(k/2)^{L-\ell}]$, where $o = (s_1-1)\cdot(k/2)^{L-1} + (s_2-1)\cdot(k/2)^{L-2} + \cdots + (s_\ell - 1)\cdot(k/2)^{L-\ell}$.

In the following, we will denote the concatenation operator by $\circ$ and call the blocks $\mathrm{B}\,(S\circ i)$, $i \in [1, k/2]$, *children* of $\mathrm{B}\,(S)$ (the block in level $\ell = 0$ has k children) and vice-versa $\mathrm{B}\,(S)$ the *parent* of the blocks $\mathrm{B}\,(S\circ i)$. Edges are only drawn between blocks that have a parent-child relationship. This can be seen in Figure 1, where the blocks are visualized as blue boxes (the block at the top is $\mathrm{B}\,(\emptyset)$). We then denote by $\mathrm{T}\,(S)$ the subgraph containing all blocks $\mathrm{B}\,(S')$ such that S is a prefix of S'. For such a fat-tree $\mathrm{T}\,(S)$, we say that it is *rooted* in $\mathrm{B}\,(S)$. Note, when compressing each block to a single node and drawing an edge for each parent-child relationship, then the resulting graph becomes a tree such that $\mathrm{B}\,(S''')$ is a successor of $\mathrm{B}\,(S'')$ iff S''' is a prefix of S''. In order to describe how edges are drawn, we also define *clusters* such that every block $\mathrm{B}\,(S)$ is partitioned into clusters. Each cluster $\mathrm{C}\,(S,i)$, $i \geq 1$, contains the first $i\cdot(k/2)$ consecutive nodes of $\mathrm{B}\,(S)$. Edges are inserted by constructing complete bipartite subgraphs. For all clusters $\mathrm{C}\,(S,i)$, we draw edges from every node in the cluster to the i-th node in each of the children of $\mathrm{B}\,(S)$ (and vice-versa). We call this set of nodes in the children *vertical cluster* $\mathrm{VC}\,(S,i)$. In Figure 2 we illustrate how these edges are drawn.

Note, from the point-of-view of a fixed node v in some level ℓ, it resides in exactly one cluster of some block $\mathrm{B}\,(S)$ with $|S| = \ell$. Furthermore, in case of $\ell > 0$, it also lies in exactly one vertical cluster, the cluster $\mathrm{VC}\,(S', i')$ where $\mathrm{B}\,(S')$ is the parent of $\mathrm{B}\,(S)$.

4.2 Routing Protocol

In the following section, we describe how the interval routing protocol of Section 3 can be adapted to the Clos topology. Note that we only consider topologies with a constant amount of layers, i.e., $L = \Theta(1) > 1$. To enable interval routing, we employ an additional layer of granularity. That is, we partition each cluster and vertical cluster into $K := (4 + L) \log k$ consecutive intervals of size $I := k/((8 + 2L) \log k) = \Theta(k/\log k)$. We denote the j-th such interval, $j \geq 0$, of each cluster $\mathrm{C}(S, i)$, and the vertical cluster $\mathrm{VC}(S, i)$ by $\mathrm{C}(S, i, j)$ or $\mathrm{VC}(S, i, j)$, respectively. A slight exception to this occurs at level 0 which only consists of a single block $\mathrm{B}(\emptyset)$. Here $\emptyset$ is used to denote the sequence of length 0. As $\mathrm{B}(\emptyset)$ has k children instead of $k/2$, there are k nodes in each cluster $\mathrm{VC}(\emptyset, i)$. These vertical clusters are also split into K many intervals, each containing $2 \cdot I$ nodes.

We focus on all-to-one routing towards some destination d which resides on level L (servers are typically located at the bottom of the Clos topology [1]). The primary tool to route packets towards d is the sequence S_d, which we define as the sequence $S \in \mathbb{S}_L$ with length $|S| = L$ that fulfills $\mathrm{B}(S) = \{d\}$. Note that for each node on level L such a sequence must exist (in Figure 1 this is visualized as in the bottom layer each node is contained in its own block). We denote by $S_d|i$, $0 \leq i \leq L$ the length i prefix sequence of S_d. Furthermore, we let $d_{i,j}$ denote the j-th node in the block $\mathrm{B}(S_d|i)$. The routing protocol follows the definitions of a local failover protocol given in Section 1.3 and equips each node with fitting distributions.

▶ **Definition 18** (Clos Interval Routing). *Let v be a node $v \in C(S, i, j)$. Protocol $\mathcal{P}_C$ equips v with the following distributions to enable routing towards d.*

(R1) *S is* not *a prefix of S_d. Let $B\left(S^P\right)$ denote the parent of v's block $B(S)$. Then $v \in VC\left(S^P, i', j'\right)$ for some $i', j' \geq 1$ and*

$$\mathcal{D}(v, \mathcal{F}_v, d) = \mathrm{Unif}\left(C\left(S^P, i', (j' + 1) \mod K\right) \setminus \mathcal{F}_v\right)$$

(R2) *S is a length-s prefix of S_d. If $d_{s+1,i} \notin \mathcal{F}_v$, set $\mathcal{D}(v, \mathcal{F}_v, d) = \mathrm{Unif}(\{d_{s+1,i}\})$. Else, set*

$$\mathcal{D}(v, \mathcal{F}_v, d) = \mathrm{Unif}\left(VC(S, i, j) \setminus \mathcal{F}_v\right)$$

The basic idea behind the routing protocol is to send a packets with destination d from child to parent blocks until they reaches a block $\mathrm{B}(S)$ such that S is prefix of S_d (R1). Assume now that, after reaching this block $\mathrm{B}(S)$, the packet lies on a node v_1 in interval $\mathrm{C}(S, i, j)$ and S is a length $|S| = s$ prefix of S_d. As $v_1 \in \mathrm{C}(S, i)$, it is connected to $d_{s+1,i}$, which lies in $\mathrm{VC}(S, i)$. After forwarding the packet to this node, it would then reside on a node in $\mathrm{B}(S_d|s + 1)$. Note that this block's sequence matches the destination for one more element. However, in case $d_{s+1,i}$ cannot be reached, the only link from v into $\mathrm{B}(S_d|s + 1)$ is unreachable. In such a case, it is forwarded to some $w_1 \in \mathrm{VC}(S, i, j)$ instead (R2). As w_1 lies on a block $\mathrm{B}(S')$, which is a child of $\mathrm{B}(S)$ that has some sequence $S' \neq S_d|s + 1$, the packet is forwarded according to R1 in the next step. Afterwards, it will again lie on a node v_2 in $\mathrm{C}(S, i)$. However, this time in the interval $\mathrm{C}(S, i, j + 1)$. In the next step, the packet is again attempted to be forwarded to $d_{s+1,i}$. Otherwise it is forwarded to $\mathrm{VC}(S', i, j + 2)$ and the procedure repeats. Intuitively, the packet "ping-pongs" between layers $|S|$ and $|S| + 1$ until it manages to reach $d_{s+1,i}$, similar as in the protocol for the complete bipartite graph of Section 3. As the forwarding partners are chosen u.a.r., it is unlikely for the packet to hit a node with failed link to $d_{s+1,i}$ in each of $\Omega(\log k)$ alternations, and it will eventually hit $d_{s+1,i}$. A visualization of this idea is given in Figure 3. We also invite the reader to familiarize her- or himself with the more detailed example we prepared in Appendix A.

▶ **Theorem 19.** *Let $G = (V, E)$ be a Clos topology with degree k and $L + 1$ levels for some constant $L > 1$. Consider the routing protocol $\mathcal{P}_C$ and assume all-to-one routing towards some destination d on level L. Assume the adversary chooses its set of failures $\mathcal{F}$ such that the following holds for every triple (S, i, j) where $S \in \mathbb{S}_L$ with $0 \leq |S| \leq L - 1$, $1 \leq i \leq L - |S|$ and $0 \leq j < K$:*

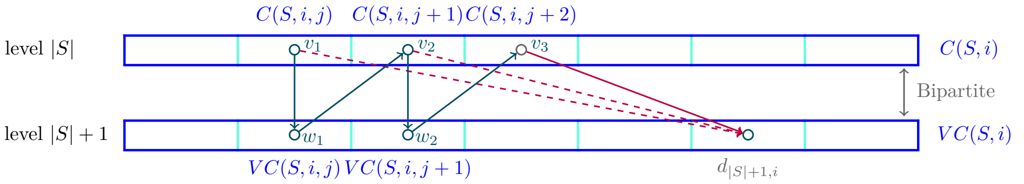


Figure 3 "Ping-Pong" of packet starting at v_1 in a block $\mathrm{B}\,(S)$ where S is prefix of S_d.

1. $\forall w \in VC(S,i) : |\{v \in C(S,i,j) \mid w \in \mathcal{F}_v\}| \leq I/3$
2. $\forall v \in C(S,i) : |\{w \in VC(S,i,j) \mid v \in \mathcal{F}_w\}| \leq I/3$

Then, with probability $1-O\left(k^{-4}\right)$, *every node* $u \in V\setminus\{d\}$ *has* $\mathcal{L}(u) = O\left(k^{L-1} \cdot \log k \log\log k\right)$, *even if the adversary knows* $\mathcal{P}_C$ *and* d.

While the requirement on the failure set $\mathcal{F}$ may seem restrictive at first, it simply states that in every interval at most $I/3 = \Theta(k/\log k)$ many nodes may have failed edges to the same node. Note that $I/3$ failures from nodes of the same interval are simultaneously allowed to many different nodes.

4.3 Analysis of Theorem 19

Throughout this proof, we consider the destination d as well as the set of failed edges placed by the adversary $\mathcal{F}$ to be fixed (we assume this set to adhere to the requirements of Theorem 19). As described in Section 1.3, each node v draws its routing entry $\alpha(v, \mathcal{F}_v, d)$ from $\mathcal{D}\,(v, \mathcal{F}_v, d)$ which is specified in Definition 18. As we consider the set of failures $\mathcal{F}$ as well as d to be fixed, we use the abbreviation $\alpha(v) := \alpha(v, \mathcal{F}_v, d)$.

Staggered Load Calculation. In the following, we will not immediately uncover all entries $\alpha(v)$ required to determine the load $\mathcal{L}(v)$ some node receives. Instead, we will uncover these entries step-by-step. To that end, we extend our notion of *load* defined in Section 1.3, to also apply in cases where some entries are still left covered. Flows that arrive at a node v with a still covered entry $\alpha(v)$, are assumed to be *stopped* and only contribute to load of nodes that lie on the path the flows takes to reach v. As soon as the entry of v is uncovered, all stopped flows continue to flow until they either reach d, or hit another node with a covered entry. It is easy to see that by increasing the number of uncovered entries, the load at any node can only increase, and, after uncovering all entries of nodes $v \neq d$, we end up with the notion of load defined in Section 1.3. This staggered uncovering of entries allows us to develop a bound on the load step-by-step and helps us circumvent dependencies of the traffic flow in different parts of the topology.

▶ **Lemma 20.** *Let* ℓ *be an integer in* $[0, L-1]$. *Then, after uncovering all entries besides those of nodes in* $T(S_d|\ell)$, *the following holds with probability* $\geq 1 - 4\ell \cdot k^{-4}$:

1. *no flow travels in a cycle, i.e.,* $\mathcal{L}(v) < \infty$ *for all nodes* v
2. *flows of nodes with uncovered entries are stopped at some node* $v \in B\,(S_d|\ell)$
3. *all nodes, including those in* $B\,(S_d|\ell)$, *have load* $O\left(k^{\ell}\right)$

Sketch of Proof. The proof uses induction over the levels $\ell = 0$ to $L-1$ in the following way. In each step of the induction we uncover the edges in $\mathrm{T}\,(S_d|\ell) \setminus \mathrm{T}\,(S_d|\ell+1)$. In the induction hypothesis, we assume that the lemma holds up to some $\ell \in [0, L-2]$ and in the induction step, we show that the statement also holds for $\ell + 1$. The base case (i.e., before we uncover any entries) trivially holds.

In order to perform the induction step, we use a two-step approach. First, we uncover the edges in $\mathcal{T}_\ell = \{\mathrm{T}\left(S_d|\ell \circ i\right) \mid 1 \leq i \leq k/2 \wedge S_d|\ell \circ i \neq S_d|\ell+1\}$ (note that if $\ell = 0$, then i is in the range $1, \ldots, k$). As an example, if $\ell = 0$ and d is the last vertex in level 3 in Figure 1, then the set $\mathcal{T}_\ell$ contains the subtrees rooted in the first three blocks of level 1. If $\ell = 1$ (d remains the same node in Figure 1), then $\mathcal{T}_\ell$ is the subtree rooted in the 7th block of level 2. After uncovering the edges in $\mathcal{T}_\ell$, we show that every vertical cluster in the blocks on level $\ell+1$ in $\mathcal{T}_\ell$ contain $O\left(k\right)$ load.

In the second step, we uncover the edges between levels ℓ and $\ell+1$ in $\mathrm{T}\left(S_d|\ell\right)$. Then, we obtain that the statement holds for $\ell+1$. This second step heavily uses the properties of the failover routing algorithm in complete bipartite graphs. The full proof is given in [3]. ◀

Proof of Theorem 19. Let $S_{L-1} := S_d|L-1$. We start with an application of Lemma 20 for $\ell = L-1$. When uncovering all entries except those in $\mathrm{T}\left(S_{L-1}\right)$, this implies that the flows of all nodes outside $\mathrm{T}\left(S_{L-1}\right)$ enter $\mathrm{T}\left(S_{L-1}\right)$ at its root $\mathrm{B}\left(S_{L-1}\right)$ without causing load higher than $O(k^{L-1})$ w.h.p. Note that the root $\mathrm{B}\left(S_{L-1}\right)$ of $\mathrm{T}\left(S_{L-1}\right)$ consists of $k/2$ nodes and therefore only a single cluster. More precisely the following holds.

▶ **Observation 21.** *$T(S_{L-1})$ is a complete bipartite graph that consists of the clusters $C(S_{L-1}, 1)$ and $VC(S_{L-1}, 1)$. These clusters each have size $k/2$ and $d \in VC(S_{L-1}, 1)$.*

This enables us to apply results from the bipartite graph section (Section 3). As S_{L_1} is a prefix of S_d, all flows starting from nodes in $\mathrm{C}\left(S_{L-1}, 1\right) \cup \mathrm{VC}\left(S_{L-1}, 1\right)$ will "ping-pong" between the clusters until d is reached (see Figure 3). Note that the path taken by the packets in $\mathrm{T}\left(S_{L-1}\right)$ according to $\mathcal{P}_C$ is exactly the same as if the nodes in $\mathrm{T}\left(S_{L-1}\right)$ would follow the bipartite routing protocol $\mathcal{P}_B$ instead (described in Section 3 with $C = (4+L)$). The main result of that section, Theorem 13, then implies that at most $O\left(\log k \cdot \log\log k\right)$ load is created w.h.p. However that result assumes that each node in $\mathrm{T}\left(S_{L-1}\right)$ starts with only 1 flow, while in our case up to $O(k^{L-1})$ flows start from a single node in $\mathrm{C}\left(S, 1\right)$ as soon as the entries in $\mathrm{T}\left(S_{L-1}\right)$ are uncovered. Thus, we obtain a maximum load of $O(k^{L-1} \cdot \log k \log\log k)$. ◀

4.4 Lower Bound for the Clos Topology

Managing load under the all-to-one traffic pattern is inherently challenging even in highly-connected Clos topologies. To illustrate this, we construct a simple congestion lower bound of $\Omega(k^{L-1})$, which holds even in the absence of link failures and does not rely on our notion of local failover routing. Assume that all-to-one routing towards a node d in level L is performed. This destination node d is incident to only $k/2$ many nodes, all of which lie in level $L-1$. All flows need to travel over one of these $k/2$ nodes to reach d. As the Clos topology contains $\Omega(k^L)$ many nodes in total and each node sends a flow towards d, it follows that one of the $k/2$ many neighbors of d must accumulate $\Omega(k^{L-1})$ flows.

In Appendix B we also present an improved lower bound that only holds for protocols that follow what we call the *faily balanced* and *shortest path* properties (see Definition 22). Among other practically relevant protocols (e.g. ECMP [16, 22]) our protocol fulfills these properties. In this setting, we show that $O\left(k/\log k\right)$ edge failures suffice for the adversary to, in expectation, accumulate $\Omega(k^{L-1}\dot{\log} k/\log\log k)$ load at some node $v \in V \setminus \{d\}$. This implies that the result in Theorem 19 is tight up to a polyloglog factor.

References

1 Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

2 Gregor Bankhamer, Robert Elsaesser, and Stefan Schmid. Local fast rerouting with low congestion: A randomized approach. In *Proc. 27th IEEE International Conference on Network Protocols (ICNP)*, 2020.

3 Gregor Bankhamer, Robert Elsässer, and Stefan Schmid. Randomized local fast rerouting for datacenter networks with almost optimal congestion, 2021. `arXiv:2108.02136`.

4 Michael Borokhovich and Stefan Schmid. How (not) to shoot in your foot with sdn local fast failover: A load-connectivity tradeoff. In *Proc. International Conference on Principles of Distributed Systems (OPODIS)*, 2013.

5 Marco Chiesa, Andrei V. Gurtov, Aleksander Madry, Slobodan Mitrovic, Ilya Nikolaevskiy, Michael Schapira, and Scott Shenker. On the resiliency of randomized routing against multiple edge failures. In *Proc. ICALP*, 2016.

6 Marco Chiesa, Andrzej Kamisinski, Jacek Rak, Gabor Retvari, and Stefan Schmid. A survey of fast-recovery mechanisms in packet-switched networks. *IEEE Communications Surveys and Tutorials (COMST)*, 2021.

7 Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrovic, Andrei Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. On the resiliency of static forwarding tables. *IEEE/ACM Transactions on Networking (TON)*, 25(2):1133–1146, 2017.

8 Marco Chiesa, Ilya Nikolaevskiy, Slobodan Mitrovic, Aurojit Panda, Andrei Gurtov, Aleksander Madry, Michael Schapira, and Scott Shenker. The quest for resilient (static) forwarding tables. In *Proc. IEEE INFOCOM*, 2016.

9 Joan Feigenbaum, Brighten Godfrey, Aurojit Panda, Michael Schapira, Scott Shenker, and Ankit Singla. Brief announcement: On the resilience of routing tables. In *Proc. ACM PODC*, 2012.

10 Klaus-Tycho Foerster, Juho Hirvonen, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. On the feasibility of perfect resilience with local fast failover. In *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

11 Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. Casa: congestion and stretch aware static fast rerouting. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 469–477. IEEE, 2019.

12 Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second igp convergence in large ip networks. *ACM SIGCOMM Computer Communication Review*, 35(3):35–44, 2005.

13 E.M. Gafni and D.P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *Trans. Commun.*, 29(1):11–18, 1981.

14 Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

15 Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.

16 Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 149–160, 2014.

17 Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

18 Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 113–126, 2013.

19 Grzegorz Malewicz, Alexander Russell, and Alexander A. Shvartsman. Distributed scheduling for disconnected cooperation. *Distributed Computing*, 18(6):409–420, 2005.

20 Mahmoud Parham, Klaus-Tycho Foerster, Petar Kosic, and Stefan Schmid. Maximally resilient replacement paths for a family of product graphs. In *Proc. OPODIS*, 2020.

21 Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. Load-optimal local fast rerouting for resilient networks. In *Proc. 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

22 Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

23 János Tapolcai. Sufficient conditions for protection routing in ip networks. *Optimization Letters*, 7(4):723–730, 2013.

24 Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM transactions on networking*, 21(2):345–358, 2012.

A Clos Topology Routing Example

In order to illustrate the behavior of our routing protocol given in Definition 18, we present an exemplary Clos topology in Figure 4. As described in Section 4.1, the nodes are partitioned into blocks, which then are again split into clusters and finally intervals. Note that, in our example, only level $\ell = 0$ has more then one cluster. Additionally, each node on level $\ell = 2$ lies in its own block. These blocks were omitted to improve visual clarity. The goal is to route the packet, currently residing at node s, towards destination node d with sequence $S_d = (3, 6)$. As described in Section 4.1, the nodes are partitioned

First (see a) and b) in Figure 4), the packet is forwarded via R1 to random nodes in an interval of a block that lies one level lower. This is done until the packet resides on a node in $\mathrm{B}(\emptyset)$. This is the first block the packet reaches such that the block's sequence (in this case $\emptyset$) is a prefix of S_d. In the next step, the packet needs to be forwarded via R2 into $T(3)$, the fat subtree containing d, to get closer to the destination d. In our example, we assume that after two hops the random choices caused the packet to land on a node v in cluster $\mathrm{C}(\emptyset, 2)$ of $\mathrm{B}(\emptyset)$. According to the definition of the Clos topology, each node inside this cluster only has a single link into $T(3)$, all of which are incident to $d_{1,2}$. In case no links are failed, the packet would simply be forwarded first to $d_{1,2}$ and then finally towards d. However, in our example we assume that the link $(v, d_{1,2})$ is failed and continue our example with Figure 5.

On the left-hand side in Figure 5 we take a closer look at $\mathrm{C}(\emptyset, 2)$ and its links into level $\ell = 1$. According to the Clos topology definition, the nodes in $\mathrm{C}(\emptyset, 2)$ and the second node in each block $\mathrm{B}(i)$ form a complete bipartite graph. These "second nodes" (denoted by $\mathrm{VC}(\emptyset, 2)$) are partitioned into vertical intervals, separated by the orange lines in the image. The idea is now to forward the packet, currently residing on v, towards $d_{1,2}$ just as in the protocol for the bipartite graph in Section 3. That is, until the packet reaches a node v' such that the link to $d_{1,2}$ is not failed, it "ping-pongs" between intervals of $\mathrm{C}(\emptyset, 2)$ and vertical intervals of $\mathrm{VC}(\emptyset, 2)$. Our routing protocol implements this as the forwarding rule applied to the packet alternates between R2 and R1.

After being forwarded to $d_{1,2}$, we continue our example with the right-hand side of Figure 5. The packet now resides in $T(3)$ and is close to the destination. If the link $(d_{1,2}, d)$ is not failed, then the packet is forwarded directly to d via R2. However, we assume that

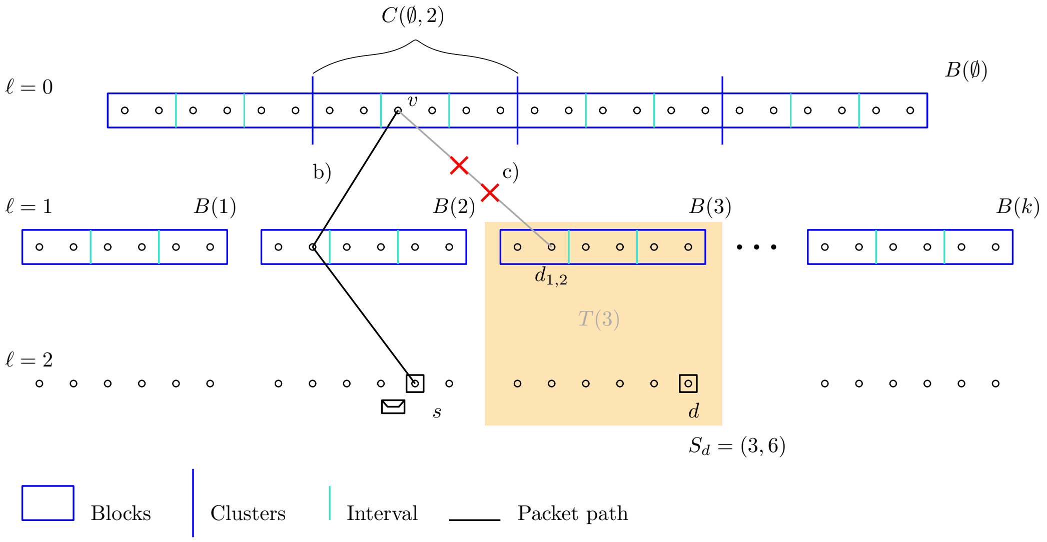


Figure 4 Clos topology with $k = 12$, $L = 2$ and interval size $I = 2$. For easier visibility, edges are not visible. A packet residing at some bottom node of layer s is forwarded towards destination d with sequence $S_d = (3, 6)$ until it encounters a failed edge.

this is not the case. Just as in subgraph considered in the left-hand side of Figure 5 one can again observe that $T(3)$ is a complete bipartite graph. This allows us to again apply ideas from the bipartite routing protocol. The packet "ping-pongs" via R1 and R2. until it hits the first node in $B(3)$ that can reach d directly.

B Refined Clos Lower Bound

In this section, we present an improved lower bound that is targeted towards a class of routing protocols that exhibit the following properties.

▶ **Definition 22** (Fairly Balanced and Shortest Path Routing). *Let $\mathcal{P}$ be a local failover protocol operating in the Clos topology, assume $\mathcal{F} = \emptyset$, i.e., no edges are failed, and assume that all-to-one routing towards any arbitrary destination d on level L is performed. We call $\mathcal{P}$ a* fairly balanced *protocol if, w.h.p., it holds that $\mathcal{L}(v) = \Theta(k^\ell)$ for $v \in B(S_d|\ell), 1 \le \ell \le L$, and $\mathcal{L}(v) = O(\text{polylog}\, k)$ otherwise.*
Furthermore, we say $\mathcal{P}$ is a shortest path *protocol, or $\mathcal{P}$* forwards over shortest paths *if the following holds for every node $v \in V \setminus \{d\}$. Let $\mathcal{F}$ be a set of failures (which might be empty) such that $|\mathcal{F}_v| \le I/3 = O(k/\log k)$. Then, the routing entry $\alpha(v, \mathcal{F}_v, d)$ must always lie on a shortest path to d in the graph $(V, E \setminus \mathcal{F}_v)$.*

While assuming these properties limits the generality of our lower bound, they are natural and realized by the standard equal-cost multipath protocol ECMP [16,22] which also underlies the widely-used routing protocols OSPF.[1] Intuitively, fairly balanced means that the load that has to enter some block at a particular level of the Clos topology is "fairly" balanced among the nodes of the particular block. That is, each such node receives the same load up to constant factors. Protocols, which do not exhibit the fairly balanced property, seem unnatural as they may generate load situations in which some nodes are heavily affected by flows while others (on the shortest path from a level 0-node to the destination) remain idle.

[1] Specifically, ECMP balances flows across shortest paths by default, and upon a failure, locally re-hashes to redistribute flows across the remaining shortest paths to the destination.

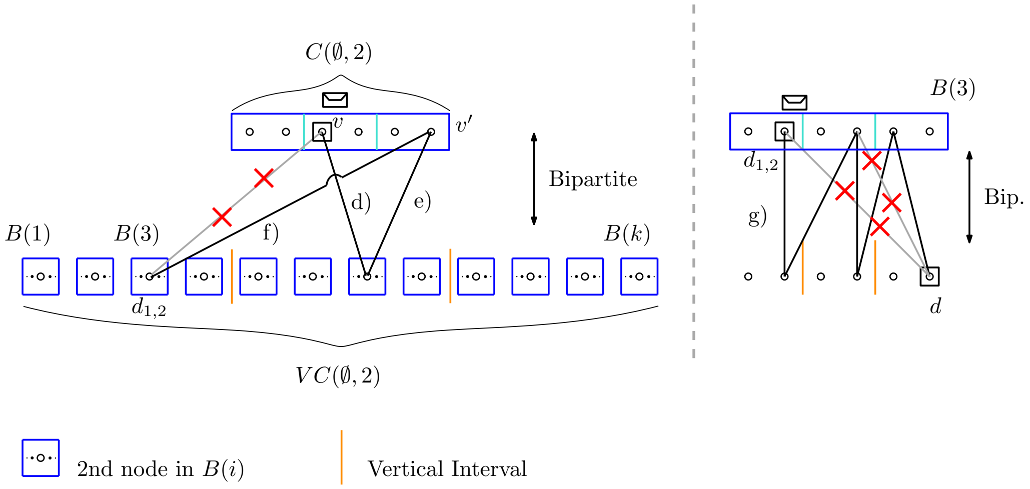


Figure 5 On the left, a closer look at C $(\emptyset, 2)$. The nodes in C $(\emptyset, 2)$ form a complete bipartite graph together with the second node in each block B (i). On the right, we have $T(3)$. After landing on $d_{1,2}$, the packet cannot be forwarded directly to d as we assume the link $(d_{1,2}, d)$ is failed.

However, note that ECMP protocol may generate cycles with probability $1/\operatorname{polylog} n$ if the number of failures is $\Omega(k/\log k)$. For protocols that exhibit above properties, one can construct a load lower bound of $\Omega(k^{L-1}\log n/\log\log n)$. To achieve this bound, the adversary only fails edges in T $(S_d|L-1)$, which is a complete bipartite graph (see Observation 21) consisting of k nodes partitioned into C $(S_d|L-1,1)$ and VC $(S_d|L-1,1)$. As we only consider fairly balanced protocols it follows that each node in C $(S_d|L-1,1)$ (the one partition of the complete bipartite graph) receives a load of $\Theta(k^{L-1})$.

Due to shortest path routing, no load will ever leave this bipartite graph again. Then, a set of edges incident to the destination can be failed such that a load of $\Omega(k^{L-1}\log k/\log\log k)$ is generated. The corresponding results can be found in Lemma 17 of Section 3.2. We then show in Lemma 24 that our routing protocol $\mathcal{P}_C$ indeed fulfills the properties in Definition 22. This implies that the result in Theorem 19 is tight. The proofs are given in the full version [3].

▶ **Lemma 23.** *Let $\mathcal{P}$ by a fairly balanced protocol that operates in a Clos topology with $L = \Theta(1) > 1$ layers and only forwards over shortest paths to some destination d on level L. Then, there exists a set of failures $\mathcal{F}$ with $|\mathcal{F}| \leq I/3 = O(k/\log k)$ such that, in expectation, at least one node $v \neq d$ has $\mathcal{L}(v) = \Omega(k^{L-1}\log k/\log\log k)$.*

▶ **Lemma 24.** *The protocol $\mathcal{P}_C$ defined in Definition 18 is a fairly balanced and shortest path protocol.*

Deterministic Logarithmic Completeness in the Distributed Sleeping Model

Leonid Barenboim ✉
The Open University of Israel, Raanana, Israel

Tzalik Maimon ✉
Ben-Gurion University of The Negev, Beer Sheva, Israel

Abstract

In this paper we provide a deterministic scheme for solving any decidable problem in the distributed *sleeping model*. The sleeping model [22, 9] is a generalization of the standard message-passing model, with an additional capability of network nodes to enter a sleeping state occasionally. As long as a vertex is in the awake state, it is similar to the standard message-passing setting. However, when a vertex is asleep it cannot receive or send messages in the network nor can it perform internal computations. On the other hand, sleeping rounds do not count towards *awake complexity*. Awake complexity is the main complexity measurement in this setting, which is the number of awake rounds a vertex spends during an execution. In this paper we devise algorithms with worst-case guarantees on the awake complexity.

We devise a deterministic scheme with awake complexity of $O(\log n)$ for solving any decidable problem in this model by constructing a structure we call *Distributed Layered Tree*. This structure turns out to be very powerful in the sleeping model, since it allows one to collect the entire graph information within a constant number of awake rounds. Moreover, we prove that our general technique cannot be improved in this model, by showing that the construction of distributed layered trees itself requires $\Omega(\log n)$ awake rounds. This is obtained by a reduction from message-complexity lower bounds, which is of independent interest. Furthermore, our scheme also works in the $\mathcal{CONGEST}$ setting where we are limited to messages of size at most $O(\log n)$ bits. This result is shown for a certain class of problems, which contains problems of great interest in the research of the distributed setting. Examples for problems we can solve under this limitation are leader election, computing exact number of edges and average degree.

Another result we obtain in this work is a deterministic scheme for solving any problem from a class of problems, denoted **O-LOCAL**, in $O(\log \Delta + \log^* n)$ awake rounds. This class contains various well-studied problems, such as MIS and $(\Delta + 1)$-vertex-coloring. Our main structure in this case is a tree as well, but is sharply different from a distributed layered tree. In particular, it is constructed in the local memory of each processor, rather than distributively. Nevertheless, it provides an efficient synchronization scheme for problems of the **O-LOCAL** class.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Computing, Sleeping Model, Complexity Class

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.10

Related Version *Full Version*: `https://arxiv.org/pdf/2108.01963.pdf` [5]

Funding This work was supported by the Open University of Israel Research Fund.

Acknowledgements The authors are grateful to the anonymous reviewers for helpful comments.

1 Introduction

What can be computed within logarithmic complexity has been one of the most fundamental questions in distributed and parallel computing, since the initiation of the study of parallel algorithms in the eighties. Various problems were shown to belong to the NC class back then, i.e., the class of problems that can be solved in polylogarithmic time by a polynomial number of machines. This includes several fundamental problems, namely, $(\Delta + 1)$-coloring, Maximal Independent Set and Maximal Matching. All of these problems admit deterministic logarithmic parallel algorithms. In the distributed setting, however, these problems turned out to be much more challenging, if one aims at a deterministic solution. The first deterministic polylogarithmic solution was an $O(\log^7 n)$ time algorithm for the problem of Maximal Matching, obtained by Hanckowiak, Karonski and Panconesi [21]. More recently, polylogarithmic deterministic $(\Delta^{1+\epsilon})$-coloring was obtained by Barenboim and Elkin [3]. The problem of $(2\Delta - 1)$-edge-coloring was provided with an $O(\log^7 \Delta \log n)$-rounds deterministic algorithm by Fischer, Ghaffari and Kuhn [16]. Recently, a plethora of results were published in this field, with various improvements to the aforementioned algorithms. See, e.g., [4, 1, 15, 23], and references therein. In a very recent breakthrough, a wide class of problems have been solved using deterministic polylogarithmic number of rounds [26], including $(\Delta + 1)$-coloring and Maximal Independent Set. This was achieved by providing an efficient algorithm for the $(O(\log n), O(\log n))$-Network-Decomposition problem, which is complete in this class.

1.1 Our Results

In the current paper we investigate yet another distributed setting, namely, the *Sleeping Setting.* Several variants of this setting have attracted the attention of researchers recently [6, 9, 11, 14, 19, 22]. The particular setting and complexity measure we consider in this paper were introduced by Chatterjee, Gmyr and Pandurangan [9] in PODC'20. This sleeping setting is similar to the standard distributed $\mathcal{LOCAL}$ setting [24], but has an additional capability, as follows. In the sleeping setting, the vertices of the network graph can decide in each round to be in one of two states; a "sleep" state or an "awake" state. If all the vertices are awake all the time, the setting is identical to the standard $\mathcal{LOCAL}$ setting. However, the capability of entering a "sleep" state is where a vertex cannot receive or send messages in the network, neither can it perform internal computations. Consequently, such rounds do not consume the resources of that vertex, and shall not be counted towards a complexity measurement that aims at optimizing resource consumption. Indeed, in this setting a main complexity measurement takes into account only awake rounds. Specifically, the worst-case awake complexity of an algorithm in the sleeping setting is the worst-case number of rounds in which any single vertex is awake. In PODC'20, Chatterjee, Gmyr and Pandurangan [9] presented a Maximal Independent Set randomized algorithm with expected awake complexity of $O(1)$. Its high-probability awake complexity is $O(\log n)$, and its worst-case awake complexity is polylogarithmic. This work raised the following two important questions:

(1) Can MIS be solved within deterministic logarithmic awake complexity?
(2) Can additional problems be solved within such complexity?

In the current paper we answer these questions in the affirmative. But much more generally, we show that *any decidable problem* can be solved within deterministic logarithmic awake complexity in the distributed sleeping setting. Namely, a decidable problem is any computational problem that has a sequential deterministic algorithm that provides a correct solution within a finite sequential running time (as large as one wishes). Note that undecidable problems in the sequential setting are also undecidable in the different variants of distributed settings. For the purpose of solving decidable problems, we present a new structure, namely, a *Distributed Layered Tree* (DLT). We show that if one is able to compute a distributed layered tree, then any decidable problem can be solved within additional awake complexity of $O(1)$. This is because a DLT allows each vertex to obtain all the information of the input graph in a constant number of awake rounds, and then any decidable problem can be solved locally and consistently by all vertices using a sequential algorithm. We also prove that DLT itself can be solved in $O(\log n)$ awake rounds. In particular, this provides a deterministic logarithmic solution to the fundamental Broadcast problem. This improves the best previously-known awake complexity of this problem in the sleeping setting, due to Chang et al. [8], by at least a quadratic factor. We note that the broadcast algorithm of Chang et al. was devised for settings with additional requirements, i.e., it is more general than Broadcast in the sleeping setting. Nevertheless, it was still the state-of-the-art even in the sleeping model. Our improvement applies specifically to the sleeping model.

A natural question is how difficult the construction of DLT is. We answer this by proving a lower bound of $\Omega(\log n)$ awake rounds for the DLT problem. This lower bound is obtained by a simple but powerful tool of a reduction from *message complexity* lower bounds in the $\mathcal{LOCAL}$ model. With this lower bound, given that the DLT problem itself is a decidable problem, we obtain a tight deterministic bound of $\Theta(\log n)$ worst-case awake time on the class of decidable DLT-hard problems [1] in the sleeping model.

An additional direction of ours is the analysis of a class we define as the **O-LOCAL** class. This is a class of problems that can be solved using an acyclic orientation of the edges, by choosing a solution for each vertex after all vertices reachable from it have computed their solution, and as a function of these solutions. A notable example is $(\Delta + 1)$-coloring, where each color can be selected once all neighbors on outgoing edges have selected their own colors, such that the color does not conflict with any of them. Another example is MIS, where each decision is made after all outgoing neighbors have made their decisions. We show that any problem that belongs to this class has deterministic worst-case awake complexity of $O(\log \Delta + \log^* n)$.

In addition to the number of awake rounds, which is the main complexity measurement in this setting, we are also interested in optimizing the overall number of communication rounds. Since the DLT can be used to solve any decidable problem, it follows that certain such problems require $\Omega(n)$ communication rounds. (These are the global problems of the ordinary distributed setting. For example, the leader election is such a problem.) We investigate how close we can get to this lower bound with an algorithm of $\tilde{O}(n)$ rounds for the distributed layered tree problem. While our basic algorithm requires $O(n^2 \log n)$ communication rounds, a more sophisticated version requires only $O(n \log n \log^* n)$ communication rounds. This comes at a price of increasing the worst-case awake complexity, but only by a factor of $O(\log^* n)$.

1.2 The Sleeping Setting

The sleeping setting represents the need for energy-efficient algorithms in ad hoc, wireless and sensor networks. In such networks, the energy consumption depends on the amount of time a vertex is actively communicating or performing calculations. More importantly, significant energy is spent even if a node is idle, but awake. It was shown in experiments that the energy consumption in an idle state is only slightly smaller than when a node is active [12, 30]. This is in contrast to the sleeping state, in which energy consumption is decreased drastically. Thus, if a node may enter a sleeping mode to save energy during the course of an algorithm, one can significantly improve the energy consumption of the network during the execution of an algorithm.

[1] A *DLT-hard* problem is a problem whose solution provides a DLT within additional $o(\log n)$ awake rounds.

The sleeping model is a formulation of this premise, and is a generalization of the traditional $\mathcal{LOCAL}$ model. In the sleeping model, similarly to the $\mathcal{LOCAL}$ model, a communication network is represented by an n-vertex graph $G = (V, E)$, where vertices represent processors, and edges represent communication links. There is a global clock, and computation proceeds in synchronous discrete rounds. In each round a vertex can be in either of the two states, “sleep” or “awake”. (While in the $\mathcal{LOCAL}$ model the vertices are only in the “awake” state.) If a vertex is in the “awake” state in a certain round, it can perform local computations as well as sending and receiving messages in that round. On the other hand, in a round of a “sleep” state, a vertex cannot send or receive messages, and messages sent to it by other vertices are lost. It also cannot perform any internal computations. A vertex decides a-priori about entering a “sleeping” state. That is, in order to enter a sleeping state in a certain round k, either the vertex decides about it in an awake round $k' < k$, or such a decision is hard-coded in the algorithm, and is known before its execution. Nodes in the “sleep” state consume almost no energy, and thus shall not be counted towards the energy efficiency analysis.

Initially, vertices know the number of vertices n, or an upper bound $\hat{n} \geq n$. The IDs of vertices are unique and belong to the set $[\hat{n}]$. Even if $\hat{n} >> n$, the awake complexity of our algorithms is not affected at all. The overall number of clock rounds, however, may be affected. In this case n should be replaced by $\hat{n}$ in the clock complexity bounds. In some of our algorithms, however, the dependency on n and $\hat{n}$ can be made as mild as the log-star function. See Section 4.

The main efficiency measures in the Sleeping Model. The measurements for the performance of an algorithm in the sleeping model were first mentioned by Chatterjee, Gmyr and Pandurangan in [9]. For a distributed algorithm with input graph $G = (V, E)$ in the sleeping model, two types of complexity measurements are defined. One is *node-averaged awake complexity* in which, for a node $v \in V(G)$, define $a(v)$ as the number of rounds v spends in the “awake” state until the end of the algorithm. The node-average awake complexity is the average $\frac{1}{n}\sum_{v\in V(G)} a(v)$.
The second efficiency measurement is the *worst-case awake complexity.* This is defined as $\max_{v\in V(G)} a(v)$ and is a stronger requirement than the node-averaged awake complexity. In this paper we focus entirely on the worst-case efficiency measurement.

1.3 Our Techniques

1.3.1 Upper Bound

Our main technical tool is the construction of a *Distributed Layered Tree.* (We denote it shortly by *DLT.*) A DLT is a rooted tree where the vertices are labeled, such that each vertex has a greater label than that of its parent, according to a given order. Moreover, in a DLT each vertex knows its own label and the label of its parent. This knowledge of the label of the parent is not trivial in the sleeping model since passing this information between parent and child requires both of them to be in an awake state. Therefore, this knowledge and hierarchy of IDs throughout the tree makes DLTs are very powerful structures in the sleeping setting. Indeed, once such a tree is computed, the information of the entire graph can be learned by all vertices within $O(1)$ awake complexity, as follows. For a non-root vertex $v \in V(G)$, let $L(v)$ and $L(p(v))$ be the labels of v and the parent of v in the DLT, respectively. Each non-root vertex $v \in V(G)$ is awake only in rounds $L(p(v))$ and $L(v)$. The root r awakes only in round $L(r)$. This way the root is able to perform a broadcast to all the vertices of the tree. Each vertex $v \in V(G)$ receives the information from the root in round $L(p(v))$ (this information has arrived to the parent of v in an earlier stage) and passes it to its children in round $L(v)$. Indeed, in this round v and all its children are awake. In a similar way, a convergecast in the DLT can be performed. We choose some label n' which is greater than all vertex labels in the DLT. Each vertex $v \in V(G)$ awakens in rounds $n' - L(p(v))$ and $n' - L(v)$. This way, information from the leaves propagates towards the root. In round $n' - L(v)$ a vertex v receives information from all its children, and in a later stage, in round $n' - L(p(v))$, the vertex forwards the information to its parent. Note that indeed $n' - L(v) < n' - L(p(v))$, since $L(v) > L(p(v))$, according to the definition of a DLT. A formal proof for the running time of the broadcast and convergecast procedures in a DLT can be found in Lemma 2.1 in Section 2.

Thus, it becomes possible to perform broadcast and convergecast in the tree within 4 awake rounds per vertex. A broadcast and convergecast in a tree allows each vertex to obtain the entire information stored in the tree. Since the tree spans the input graph, the entire information of the graph is obtained. Then, any decidable problem can be solved using the same deterministic algorithm internally in all vertices of the graph. Finally, each vertex deduces its part in the solution. This execution that is performed internally, does not require any additional rounds of distributed communication and is considered as a single awake round in terms of the sleeping model. To summarize this discussion, a DLT makes it possible to solve any decidable problem within 5 awake rounds per vertex.

The ability to solve any decidable problem within a constant awake complexity suggests that the computation of a DLT is an ultimate goal in the sleeping setting. Thus establishing the efficiency of this construction is of great interest. We construct a DLT within $O(\log n)$ awake rounds as follows. We begin with n singleton DLTs, where each vertex of the input graph is a DLT in a trivial way. Then, we perform $O(\log n)$ connection phases in which the trees are merged. Each phase requires at most $O(1)$ awake rounds from each vertex. The number of DLT trees in each phase is at least halved. After $O(\log n)$ phases, a single tree remains. This DLT contains all the vertices of the input graph. Thus, it is the DLT of the entire input graph G.

The high-level idea of our algorithm is somewhat similar to the celebrated algorithm of Gallager, Humblet and Spira for minimum spanning trees [18]. But the construction is fundamentally different. While GHS finds an *existing* subgraph that is an MST, our technique gradually builds trees that contain *new data*. These are new ID assignments that make it possible to have progress with trees formation. In each iteration trees are merged and IDs are reassigned, until a single DLT of the entire input graph is achieved. This tree has the desired IDs, according to the definition of the DLT, as a result of the ID recomputation made in each iteration.

1.3.2 Lower Bound

Once we establish an upper bound on the awake complexity of DLT, we turn to examining lower bounds. We note that an ordinary lower bound technique may not work for the sleeping setting. This is because the standard techniques in the distributed setting deal with what information can be obtained within a certain number of rounds. That is, within r rounds, for an integer r, each vertex can learn its r-hop neighborhood. Then arguments of indistinguishably of views are used. (That is, vertices that must make distinct decisions are unable to do that, if their r-hop-neighborhoods are identical. In this case, r rounds are not sufficient to solve a certain problem.) However, such arguments do not work in the sleeping setting. Indeed, within $O(1)$ awake rounds the entire graph can be learned on certain occasions. Thus, algorithms with r awake rounds are not limited to obtaining knowledge of r-hop-neighborhoods.

As a consequence of the latter phenomenon, we investigate alternative ways to prove lower bounds. We introduce a quite powerful technique that allows one to transfer lower bounds on message complexity into lower bounds for rounds in the sleeping setting. Indeed, if tn messages must be sent in a ring network to solve a certain problem, for an integer $t > 1$, then $t/2$ awake rounds are required for any algorithm that solves the problem in the sleeping setting. Otherwise, if $t' < t/2$ awake rounds are possible, all the messages in each round can be concatenated, and thus each vertex sends up to t' messages to each of its two neighbors in the ring, during the t' rounds it is awake. The number of messages per vertex becomes $2t' < t$, and the overall number of messages passed is thus smaller than tn, which contradicts the assumption that at least tn messages must be sent. A formal proof for this claim can be found in Lemma 3.1 in Section 3.

We employ this idea with the known lower bound of $\Omega(n \log n)$ for message complexity of leader election in rings [17]. Since a DLT allows, in particular, to solve leader election, we deduce that $\Omega(\log n)$ awake rounds are required. Otherwise, it would be possible to solve leader election in rings within fewer than $\Theta(n \log n)$ messages. We note that the lower bound of [17] holds for a given number of rounds assuming the IDs are sufficiently large. Our upper bounds on awake complexity, on the other hand, do not rely on the range of IDs, but only on the number of vertices in the graph. No matter how large the IDs are, on an n-vertex graph the awake complexity for constructing a DLT is $O(\log n)$. The overall number of clock rounds (awake and asleep) do depend on the range of identifiers, but the dependency can be made as low as the log-star function, by using the coloring algorithm of Linial [24]. Our algorithm is applicable also with proper coloring of components, not necessarily with distinct IDs. Consequently, for any given number of clock rounds (awake and asleep), which upper bounds the ordinary running time of an algorithm, there exists a sufficiently large range of IDs, such that the awake complexity $O(\log n)$ of our algorithm is tight.

1.3.3 Improved Upper-Bound for O-LOCAL problems

O-LOCAL problems are those that can be solved sequentially, according to an acyclic orientation provided with the input graph, such that each vertex decision is made after all vertices emanating from it have made their own decisions, and as a function of these decisions. (Note that directing edges from endpoints of smaller IDs to larger IDs provides such an orientation.) For these kind of problems, we employ a technique that is quite different from that of a DLT, and obtain awake complexity of $O(\log \Delta + \log^* n)$. We still employ a tree construction, but this time it is more sophisticated than a DLT. On the other hand, it is constructed internally by each vertex, and is the same in all vertices. The algorithm starts by a distributed computation of an $O(\Delta^2)$-coloring of the input graph within $O(\log^* n)$ time. Then, each vertex constructs internally a binary search tree, whose leaves are the possible $O(\Delta^2)$-colors. (The colors are not consecutive, and inner nodes have integer values between the values of the colors.) Next, each vertex decides to wake-up in the rounds whose numbers appear in the path from the leaf of their color and the root. We prove that one of these rounds occurs after all neighbors of smaller colors have made their decisions. Moreover, by that round these vertices have communicated their decision to the vertex. Consequently, it may compute its own decision. Since the depth of the tree is $O(\log \Delta)$, this requires only $O(\log \Delta)$ awake-rounds per vertex.

1.4 Related Work

The distributed $\mathcal{LOCAL}$ model was formalized by Linial in his seminal paper [24] from 1987. This paper also provides a deterministic $O(\Delta^2)$-coloring algorithm with $O(\log^* n)$ round-complexity, as well as a matching lower bound. Since then, a plethora of distributed graph algorithms has been obtained in numerous works. See, e.g., the survey of [27] and references therein.

The sleeping setting has been intensively studied in the area of Computer Networks [10, 25, 28, 29]. In Distributed Computing the problem of broadcast in sleeping radio networks was studied by King, Phillips, Saya and Young [22]. The problem of clock synchronization in networks with sleeping processors and variable initial wake-up times was studied by Bradonjic, Kohler and Ostrovsky [7], and by Barenboim, Dolev and Ostrovsky [2]. A special type of the sleeping model, in which processors are initially awake, and eventually enter a permanent sleeping state, was formalized by Feuilloley [13, 14]. An important efficiency measurement in this setting is the *vertex-averaged* awake complexity. This setting was further studied by Barenboim and Tzur [6], who obtained various symmetry-breaking algorithms with improved vertex-averaged complexity.

The awake complexity of various problems has been also studied in radio models of general graphs (rather than unit disk graphs). In particular, several important results were achieved by Chang et al. in PODC'18 [8]. That work considered Broadcast and related problems in several radio models, that can be seen as the sleeping model with additional restrictions. Specifically, in the model that is the closest to the sleeping model, the vertices are able to either transmit or listen in an awake round, but not both. (In other words, this is a half-duplex communication model, while the sleeping model is full-duplex.) There are also even more restricted models studied in [8], in which vertices cannot receive messages from multiple neighbors in parallel.

Since the results of Broadcast [8] are applicable to the sleeping model, and they are the state-of-the-art even in this model that we consider in the current paper, a comparison between them and our results is in place. The Broadcast algorithm of [8] with the best deterministic awake complexity has efficiency $O(\log n \cdot \log N)$, where $N \geq n$ is the largest identifier. Our results, on the other hand, provide a deterministic Broadcast algorithm in the sleeping setting with awake complexity of $O(\log n)$. This is at least a quadratic improvement in the sleeping setting. We stress that our awake complexity is not affected by the size of identifiers, and remains $O(\log n)$, no matter how large N is. The Broadcast algorithm of [8] constructs trees that partition the graph into layers, but these trees are very different from our DLTs, both in their structure and in the techniques for achieving them. Specifically, in [8] each vertex in layer $i = 1, 2, ...$ in the tree has a neighbor of layer $i - 1$. On the other hand, a DLT does not necessarily have this property (This is because layer i in a DLT consists of all vertices labeled by i in the tree, which are not necessarily at distance i from the root.) In addition, the tree construction in [8] is based on ruling sets, while our techniques are considerably different.

The class **O-LOCAL** of problems that we mentioned in Section 1.3.3 is inspired by the class **P-SLOCAL** which was first defined by Ghaffari, Kuhn and Maus [20]. This class consists of all problems that can be solved as follows. Given an acyclic orientation, the output of each vertex is determined sequentially, according to the orientation, after vertices on outgoing edges have made their decisions. The decision of each vertex is based on information from a polylogarithmic radius around it. The **O-LOCAL** class is similar to the **P-SLOCAL** class, except that instead of examining a polylogarithmic-radius neighborhood around a vertex, its neighbors on outgoing edges are examined. (And, more generally, all vertices on consistently oriented paths emanating from a vertex are inspected.)

2 Distributed Layered Trees

In this section we describe our method with which one can solve any decidable distributed problem in the sleeping model. We describe the construction of a certain kind of a spanning tree, called a *Distributed Layered Tree*, defined as follows. Each vertex v in the tree is labeled with a label $L(v)$. These labels must have some predefined order, such that they can be mapped to natural numbers. The labeling is such that the label of each vertex, besides the root, is larger than the label of its parent, and each vertex knows the label of its parent. These two requirements of the spanning tree allow us to perform broadcast across the spanning tree in a fashion where each vertex is awake for exactly 2 rounds. This is also true for a convergecast procedure. Consequently, given such a tree, the root can learn the entire input graph, compute a solution for any decidable problem internally, and broadcast it to all vertices, all within a constant number of awake rounds. This is done in the following way. For a broadcast procedure, we start with a message from the root of the tree through the tree, where each vertex v is awake for just 2 rounds. Namely, v awakes in round $L(v)$ and round $L(p(v))$, where $p(v)$ is the parent of v in the spanning tree. In other rounds v is asleep. This ensures that a message sent from the root will propagate in the tree and eventually arrive to all vertices of the graph while having each vertex awaken exactly twice. In the same manner we perform the convergecast procedure, where each child v sends its message to its parent at the round $n' - L(p(v))$ for some $n' \geq n$. Again we have each vertex v active for only two rounds, specifically, $n' - L(v)$ and $n' - L(p(v))$.

Using this method one can have the root of the spanning tree compute the solution for any decidable problem $\mathcal{P}$ deterministically and broadcast this solution back through the tree to all the vertices in the input graph G with $O(1)$ worst-case complexity in the sleeping model. Therefore, our main goal is obtaining a distributed sleeping algorithm for computing such a layered tree. We begin with a formal definition of notations and the definition of a Distributed Layered Tree. Note that we refer here to lexicographic order. For sake of definition we do not limit or define this lexicographic order since it does not matter for the definition of a DLT. One can build a DLT with any order that can be mapped to natural numbers. We define a lexicographic order that serves our purposes in Section 2.1.

Vertex Label $L(v)$. The label of a vertex v is denoted by $L(v)$. The labels are taken from a range of labels which has a lexicographic order.

Tree Label $L(\mathcal{T})$. The label of a tree $\mathcal{T}$ is denoted by $L(\mathcal{T})$. The labels are taken from a range of labels which has a lexicographic order. The label of a tree is defined to be the label of the root of the tree. Hence, that label can always be found in the memory of the root.

▶ **Definition 2.1** (A Distributed Layered Tree (DLT).)**.** A DLT is a rooted oriented labeled spanning tree with two properties, with respect to some lexicographical order:

1. For each vertex v and a parent $p(v)$, the label of v is greater than the label of $p(v)$ in the lexicographical order.
2. v has knowledge of the label of $p(v)$.

As we show in the next lemma, DLTs are useful for distributing data across the graph in an efficient way.

▶ **Lemma 2.1.** *Given a DLT, the procedures of broadcast and convergecast across the entire tree take exactly 2 rounds each in the sleeping model.*

Proof.

Broadcast. Each vertex v is awake in rounds $L(v)$ and $L(p(v))$. As a vertex v awakes in round $L(v)$ and broadcasts a message, each of its children u in the tree are awake in round $L(p(u)) = L(v)$ and thus can receive the message from its parent. Therefore, a message sent by the root propagates throughout the tree until it reaches all leaves.

Convergecast. Let n' be an integer, such that $n' > L(v)$, for all $v \in V$, and n' is known to all vertices. Each vertex v is awake in rounds $n' - L(v)$ and $n' - L(p(v))$. If a child v has a message to pass to its parent $p(v)$, it awaits round $n' - L(p(v))$ and then v sends the message to $p(v)$. In that round $p(v)$ is awake and ready to receive the messages from all its children. Each vertex v already has knowledge of the subtree rooted at it, since for each vertex u in the subtree in which v is the root, $L(u) > L(v) > L(p(v))$ and thus the round $n' - L(u)$ comes before the round $n' - L(p(v))$. Thus the message propagates up the tree all the way to the root. ◀

We note that in the proof above we require for an integer n' to be larger than all labels of all vertices in the tree. Since the labels are required to be taken from a range with lexicographic order, and we have knowledge of the size of this range, n' can be chosen appropriately. We describe the lexicographic order in Section 2.1, as well as describing how each vertex has knowledge of ranges from which labels are selected.

2.1 The Connection Phases

Our algorithm for constructing a DLT starts with a graph where each vertex is considered to be a singleton tree. The initial label of each such tree $\mathcal{T}_v$, $v \in V$, is determined by the ID of its vertex v as follows. $L(\mathcal{T}_v) = ID(v)$. The vertex label is set as $L(v) = \langle ID(v), 0\rangle$. (Two coordinates are used, since trees are going to be merged and have many vertices. Then the left-hand coordinate is going to be the same for all vertices in a tree, while the right-hand coordinates may differ. Also, distinct trees will have distinct left-hand coordinate.) Each of these n singleton trees is a DLT in a trivial way. Our goal is merging these trees in stages, so that eventually a single DLT remains that spans the entire input graph. Assume we have a forest of DLTs. Initially, we have a forest of n single-vertex trees. During the connection phases, we enforce a rule regarding the representation of the labels of the vertices. Let $\mathcal{T}$ be a tree in our forest. The DLT label of $\mathcal{T}$, $L(\mathcal{T})$ is an integer number. The label of each vertex $v \in \mathcal{T}$ is set as $\langle L(\mathcal{T}), l\rangle$, where l is a number assigned to v, as described in Section 2.1.1. In what follows we describe the ordering of the vertex labels. The left coordinate is considered to be the more significant one among the two. That is, $\langle a, b\rangle < \langle c, d\rangle$ iff $a < c$ or ($a = c$ and $b < d$). Note that the requirements on labels hold. That is, the labels have an ordering and the root of the tree can deduce the tree label from its own label by referring to the first coordinate. Next, we describe how our algorithm produces connections between the trees. We do this in two stages.

2.1.1 Connection Stage One – Several DLTs into a single DLT

In this stage our goal, for each tree $\mathcal{T}$, is finding an edge (u, w) that connects $\mathcal{T}$ to a neighbor DLT $\hat{\mathcal{T}}$, such that $L(\mathcal{T}) > L(\hat{\mathcal{T}})$. In this sense, $u \in \mathcal{T}$ and $w \in \hat{\mathcal{T}}$. Using this edge we connect $\mathcal{T}$ with $\hat{\mathcal{T}}$, such that w becomes the parent of u. In the case that $\mathcal{T}$ is a single vertex v, we simply choose the neighbor of v in G with a label smaller than that of v. We note that there may be a case that no such edge is found, since $\mathcal{T}$ is a DLT with local minimum label. We handle such a case in Section 2.1.2. If $\mathcal{T}$ contains more than one vertex, we perform a convergecast from all the neighbors of all vertices in $\mathcal{T}$ to the root of $\mathcal{T}$. Consequently, the root of $\mathcal{T}$ learns the structure of $\mathcal{T}$ and the set of edges that connect $\mathcal{T}$ with other trees.

The root chooses the edge (u, w) which connects $\mathcal{T}$ to a vertex $w \in \hat{\mathcal{T}}$, where $\hat{\mathcal{T}} \neq \mathcal{T}$ is a neighboring DLT of $\mathcal{T}$. As explained above, the choice is made such that $L(\hat{\mathcal{T}}) < L(\mathcal{T})$. Recall that at this point $L(\hat{\mathcal{T}})$ appears in the first coordinate of all the labels of the vertices in $\hat{\mathcal{T}}$, and hence the root of $\mathcal{T}$ has knowledge of all the labels of its neighboring DLTs.

Internally, the root of $\mathcal{T}$ calculates a new label arrangement of the vertices of $\mathcal{T}$, such that the vertex u that was chosen above (that is connected to $w \in \hat{\mathcal{T}}$) becomes the new root, and $\mathcal{T}$ remains a DLT, under the new label arrangement. This requires a new orientation of $\mathcal{T}$. This new oriented tree is denoted $\mathcal{T}'$. The label arrangement of $\mathcal{T}'$ is obtained in the following way. Note that the DLT label of $\mathcal{T}$ is $L(\mathcal{T})$. The new root u sets its label to be $\langle L(\mathcal{T}), 0\rangle$. The rest of the vertices in $\mathcal{T}'$ are assigned labels in the following way. Each vertex v is assigned the label $\langle L(\mathcal{T}), l\rangle$ where l is the distance of v from u in $\mathcal{T}'$. Specifically, l is the level of v in $\mathcal{T}'$. It follows that each level of $\mathcal{T}'$ has labels smaller than the labels in the next level of the tree. Once this internal computation is done, $r_{\mathcal{T}}$ sends the resulting label arrangement in a broadcast procedure to all vertices in $\mathcal{T}'$. Note that u becomes the new root, instead of $r_{\mathcal{T}}$, from now until further notice.

But now we are posed with the problem that neighbors of the vertices of the tree $\mathcal{T}'$ do not know the labels of their neighbors, which is required for the next connection phase as well as the second requirement for a DLT. This is solved by awakening the entire graph for one round. Each vertex sends its knowledge to all of its neighbors while receiving the knowledge from all of its neighbors in the same round. Then all vertices of the graph switch to asleep status. The round in which this happens is determined by all vertices before the beginning of the execution. Once all vertices determine the starting time of such a phase, they all wake up after cn' rounds, for a constant c, such that cn' bounds from above the execution time of this phase. In the following lemma we prove that this connection procedure connects several DLTs into one DLT. The proof is available in the full version of this paper [5].

▶ **Lemma 2.2.** *Let C be a connected component in our graph produced at this stage. Then C is a DLT.*

We would like to note here that at the end of each connection phase each DLT C'' in the new forest has a distinct DLT label. This is due to the fact that the DLT label of C'' is set from one of the labels of the trees composing C''. If each DLT had a distinct label at the start of the connection phase, it is clear we preserved this property also once the connection phase is done. Thus, it is also true in the start of the next connection phase. Note that we start the algorithm with each DLT having a distinct DLT label since we set those labels according to the IDs of the vertices of G.

2.1.2 Connection Stage Two – Connecting Local Minimums

We now make a second connection step as part of the connection phase. The motivation in this step is that there might be trees where the DLT labels are local minima and thus those trees did not connect to any other tree. They might also not have been chosen by any other tree. If such is the case, these trees made it through stage one of the connection phase without connecting to any other component.

We would like to at least halve the number of trees in each phase and have at most $O(\log n)$ connection phases, we connect these local-minimum label trees to other components. Let T_m be such a local-minimum label tree. First, we would like to verify that T_m indeed has no connections to other components from the previous stage. If some other component has selected T_m earlier, we no longer need to handle it, even though T_m is a minimum label tree. We only need to handle trees with no connection to other trees. We check this by performing a convergecast in each connected component sending signals from the leaves to the root. If r_{T_m} received a signal from vertices that previously did not belong to T_m then T_m is not a problematic tree and was chosen by another tree in the first stage. This signal can be, for example, the label of a leaf. Since the label contains the root of the tree to which the leaf belongs (before the trees are merged), then r_{T_m} can deduce that the leaf was not part of T_m before the first stage of the connection phase. In this case we do nothing with T_m.

The other option is where T_m was not connected to any other component. In this case, we add such a connection. To this end, r_{T_m} chooses an edge (v, u) to connect to a DLT C arbitrarily. Since it is the only edge connecting it to other DLTs, the result is a tree. A simple move now would be to make C a subtree of the DLT T_m. But we note that T_m might not be the only tree that arbitrarily chose to connect to C. Since there can be more than one local minimum DLT that chooses C, we need to make C the parent of all those DLTs which chose to connect to it. Otherwise, C will become the child-DLT of more than one DLT which breaks the structure of a directed tree we aspire for.

Therefore, in order for the result to be a DLT we need to make T_m the sub-DLT of C instead. That is, given that (v, u) is the edge connecting T_m to C, we turn v into the root of T_m and make v the child of u in $C \cup T_m$. Doing so poses a problem since $L(v) < L(u)$ and this violates the requirements for a DLT in the resulting component. We solve this by waking up the entire graph for a single round and have v and u exchange information.

After this round, the information about the local-minimum label trees that asked to join C is located in vertices of the component C. This information is then delivered to the root of C, r_C by a convergecast procedure. r_C performs label reassignment in the same way as in Section 2.1.1. Specifically, a single BFS is computed for all vertices in C and the local minimum label trees T_m that connected to C. Then reassignment of these labels is made according to the levels of the BFS tree. This results in labels of the from $\langle ID(r_C), l\rangle$, such that the first coordinate is the same for all vertices in the connected component and l is the level of the vertex in C. Note that the structure of the labels and the label arrangement is the same as described in Section 2.1.1. Thus, the proof of Lemma 2.2 applies here for the new connected component and its labeling. Therefore, C is a DLT. This completes the description of connection stages. See Appendix C for a pseudocode with a summary of their steps. In what follows, we analyze the algorithm.

2.2 Analysis

We turn to analyse our algorithm for spanning a DLT on the input graph G. We prove several lemmas and conclude with the main result for our scheme. The proof is available in the full version of this paper [5].

▶ **Lemma 2.3.** *Let C be a connected component at the end of a connection phase. Then C is a DLT.*

Next, we analyze the awake complexity of the algorithm. Our claim is that a connection phase halves the number of DLTs in the forest. This is quite straightforward. If a DLT is not connected in stage one of the connection phase, phase two considers it as problematic and makes sure it connects to another DLT. Thus, every DLT connects to another DLT and thus the number of DLTs is at least halved.

The next lemma analyses the performance of a connection phase. The proof is available in the full version of this paper [5].

▶ **Lemma 2.4.** *Each vertex v is awake for at most $O(1)$ rounds in each connection phase.*

We can now conclude the analysis of our DLT spanning algorithm. Since the number of DLTs is at least halved in each phase, there are $O(\log n)$ such phases, and the following theorem is directly obtained from Lemma 2.4.

▶ **Theorem 2.5.** *A DLT for any input graph G can be deterministically computed in $O(\log n)$ awake rounds in the sleeping model.*

As shown in Lemma 2.1, the action of convergecast and the action of broadcast on the resulting spanning tree each require only $O(1)$ time in the sleeping model and thus we can collect the topology of the entire input graph to the root of our spanning tree, calculate the solution to the problem $\mathcal{P}$ deterministically and broadcast the solution to all vertices through our spanning tree, again in $O(1)$ time. This places an upper bound on the class of all decidable problems as we conclude in the following theorem.

▶ **Theorem 2.6.** *Any decidable problem can be solved within $O(\log n)$ worst-case deterministic awake-complexity in the sleeping model.*

3 A Tight Bound for DLT

In this section we prove that the complexity of DLT in the sleeping model is $\Omega(\log n)$. The proof is by a reduction from leader election on rings. For the latter problem, it is known that a certain number of messages must be sent in the network in order to solve it [17]. In what follows we prove that this lower bound on messages implies a lower bound on awake rounds in the sleeping model. Before presenting the proof, we need the following lemma, which demonstrates a connection between the number of messages an algorithm produces and the complexity in the sleeping model.

▶ **Lemma 3.1.** *Any algorithm $\mathcal{A}$ which requires at least $\Omega(\Delta n \log n)$ messages for its execution has an awake complexity of at least $\Omega(\log n)$ in the sleeping model.*

Proof. Let the number of messages that must be sent during the execution of $\mathcal{A}$ be $c\Delta \cdot n \log n$ where $c > 0$ is a constant. We show that there is at least one vertex v that must be awake for at least $c \log n$ rounds. Assume for contradiction that all vertices in G are awake for less than $c \log n$ rounds. Each vertex sends at most Δ messages (one across each adjacent edge) in a single awake round. If more than one message per edge per round is required, all these messages can be concatenated into a single message. Thus, each vertex sends less than $\Delta \cdot c \log n$ messages, and the overall number of messages in the execution is less than $n(\Delta \cdot c \log n) = c\Delta \cdot n \log n$. This is a contradiction. Therefore, there must be a vertex that is awake for $\Omega(\log n)$ rounds. We conclude that the awake round complexity of $\mathcal{A}$ in the sleeping model is also $\Omega(\log n)$. Given that there are at least $\Omega(\Delta n \log n)$ messages and n vertices and at most Δn edges, on average, each vertex is awake for at least $c \log n$ rounds. Thus, the running time of $\mathcal{A}$ (in the worst case and average case) is at least $\Omega(\log n)$. ◀

▶ Remark. An algorithm $\mathcal{A}$ that requires $\Omega(\Delta n \log n)$ messages has an awake complexity of $\Omega(\log n)$, not only in the worst vertex, but also on average over the vertices. (Such an average complexity is referred to as *vertex-averaged complexity* [9].) Indeed , if the vertex-averaged awake complexity is $o(\log n)$ then the sum of awake rounds for all vertices is $o(n \log n)$, and the number of messages is $o(\Delta n \log n)$, according to the proof of Lemma 3.1.

Next, we employ Lemma 3.1 in order to prove that DLT requires $\Omega(\log n)$ complexity in the sleeping model. We show this for a ring graph by a reduction from the leader election problem.

▶ **Theorem 3.2.** *Let $t > 0$ be an arbitrarily large integer, and $\mathcal{A}$ any deterministic algorithm for the DLT problem, which requires t rounds in the $\mathcal{LOCAL}$ model. Then there is an ID assignment from a sufficiently large range, as a function of t, such that $\mathcal{A}$ requires $\Omega(\log n)$ awake-complexity in the sleeping model.*

Proof. The proof is by contradiction. Assume that there is an algorithm $\mathcal{A}$ with awake-round complexity of $o(\log n)$, overall complexity $t > 0$, for ID assignment from an arbitrarily large range. Then $\mathcal{A}$ uses at most $o(n\Delta \log n)$ messages (see Lemma 3.1). Let C be an n-vertex cycle graph. The maximum degree of C is $\Delta = 2$. We execute $\mathcal{A}$ on C in the ordinary (not-sleeping) $\mathcal{LOCAL}$ model. We obtain a DLT of C within t rounds. Now, the root can be elected as the leader, and the other vertices know that they are not the root. In a DLT they also know the ID of the root. Thus, we have an algorithm for leader election in the $\mathcal{LOCAL}$ model which employs at most $o(n \log n)$ messages.

According to [17], the leader election problem requires $\Omega(n \log n)$ messages, if vertex IDs are chosen from a set of sufficiently large size $R(n, t)$, where R is the Ramsey function and t is the running time of the algorithm. This is a contradiction. ◀

It follows that any problem whose solution can be used to elect a leader within $o(\log n)$ additional awake rounds requires $\Omega(\log n)$ awake-complexity. We denote the class of such problems by **DLT-hard** problems. Theorems 2.5 and 3.2 directly give rise to the following corollary.

▶ **Theorem 3.3.** *The class of DLT-hard problems has a deterministic complexity tight bound of $\Theta(\log n)$ in the sleeping model.*

An alternative proof. The following alternative proof was suggested by an anonymous referee. A lower bound of [8] shows that broadcast on a path requires $\Omega(\log n)$ awake rounds. Their model allows each vertex either to transmit or listen in a round. However, the proof goes through even when vertices may transmit and listen in the same time. Consequently, a DLT requires $\Omega(\log n)$ awake rounds as well. The lower bound proof of [8] applies also to randomized algorithms.

▶ Remark. Note that in Section 2.1 we showed that the DLT problem is complete in the class of decidable problems and Theorem 3.3 states it is $\Omega(\log n)$-hard under this class.

4 Solving Oriented-Local Problems

In this section we devise an algorithm for solving a class of *Oriented-Local* problems. This class contains all problems which, given an acyclic orientation on the edge set of the graph, can be solved as follows. Each vertex awaits all neighbors on outgoing edges to produce an output, and then computes its own output as a function of the outputs of these neighbors. (Vertices with no outgoing edges produce an output immediately.) We define this class formally.

▶ **Definition 4.1.** The class of **1-hop Oriented Local Problems (1-O-LOCAL)** consists of all problems that, given an acyclic orientation μ on the edge set of G, can be solved in the following way. Let v be a vertex in G. Let U be the set of neighbors of v in its 1-hop neighborhood which precedes v in the orientation μ, i.e., the vertices connected by outgoing edges from v. Let $s(p)$, for $p \in U$, be the solution of the problem. Then, v can internally calculate $s(v)$ with the knowledge of $\{s(p) \mid p \in U\}$.

The class of **Oriented Local Problems (O-LOCAL)** is a generalization of 1-O-LOCAL, where the set U contains all vertices on paths that emanate from v, rather than v's immediate neighbors on such paths.

As one can tell, a solution for a problem in this class depends on a given orientation. Such orientation can be calculated or given as an input to the algorithm. In this work we assume that no orientation is given and we are forced to calculate one as part of the solution. We note that MIS and $(\Delta + 1)$-vertex-coloring are examples of well-studied problems which fall in the class of **1-O-LOCAL** problems.

We start with an algorithm for $O(\Delta^2)$-vertex-coloring in $O(\log^* n)$ time [24]. This gives us an orientation of the edges where we orient edges in descending order, i.e., each edge is oriented towards the endpoint of a smaller color. We have all vertices in G awake during the entire coloring algorithm. Let $q = O(\Delta^2)$ be an upper bound on the number of colors of the algorithm of Linial such that q is a power of 2. At the next stage each vertex v builds a binary search tree internally. The size of the tree is $2q - 1$. The root of the tree receives the label in the middle of the range $[2q - 1]$, which is q. Now we have $q - 1$ values in each side of the tree. Specifically, $\{1, \ldots, q - 1\}$ for the left subtree and $\{q + 1, \ldots, 2q - 1\}$ for the right subtree. We choose the middle of the range $\{1, \ldots, q - 1\}$ for the left child of the root and the middle of the range $\{q + 1, \ldots, 2q - 1\}$ for the right child of the root. We continue this recursively, so that each node of the tree obtains a unique value from $[2q - 1]$.

Now we recolor the vertices of the input graph using the following mapping. The recoloring is performed by all vertices in parallel, with no communication whatsoever. We map the elements from $[q]$ to the set of values appearing in the leaves of the binary tree. The mapping is the same in all vertices. Specifically, for each $i \in [q]$, the i-th element is mapped to the label of the i-th left-most leaf of the tree. Consequently, all vertices that were initially colored by the color i switch their color to the label of the $i$the leaf in the tree. Note that each pair of neighbors select distinct leaves of the tree, since their original colors are distinct. Therefore, the coloring after the mapping is proper as well.

Next, we switch to the sleeping state for all the vertices in the graph, and start solving a given **1-O-LOCAL** problem P. For the sake of simplicity, we proceed with the problem of MIS, but our method can be applied to any **1-O-LOCAL** problem, as would be obvious from the description of the algorithm. The scheme is as follows. Each vertex v employs its color in the $O(\Delta^2)$-coloring, and a respective leaf in the binary tree, whose value equals the color of v. Let R be the path from the leaf of the color of v to the root of the binary tree. Let $r(v) = \{r_1, \ldots, r_k\}$ be the values appearing in R. We denote $r' = r_1$. Note that some values in $r(v)$ may be greater than r', while other values may be smaller than r'. Then v awakes at each round $r_i \in r(v)$, and sends a message to its awake neighbors about its state, e.g., whether it is in the MIS, not in the MIS or undecided. It also receives such messages from its awake neighbors in these rounds. Recall that $r' = r_1$ is the round number that is equal to the color of v. In round r' the vertex v makes a decision if to join the MIS or not according to the information received from outgoing edges. The neighbors on such edges have smaller colors, and thus have made a decision before round r'. See Appendix A for an example of a colored binary tree. The following lemma proves that v has all the information from vertices of lower colors when round r' arrives. See the proof in Appendix B.

▶ **Lemma 4.1.** *At round r', which is mapped to the color of v, all vertices with colors smaller than that of v have already made a decision. Furthermore, their decisions have been passed to v in a previous round.*

For a problem P in **1-O-LOCAL**, a vertex v can make its decision in round r'. For example, the following decisions are made in some well-studied **1-O-LOCAL** problems: For MIS, v joins the MIS if all neighbors with lower colors are not in the MIS. For $(\Delta+1)$-vertex-coloring, v chooses a new color from the palette $[\Delta + 1]$ which is not yet chosen as a new color by its neighbors with lower old-colors (i.e., colors according to the initial orientation).

The depth of a binary tree with $O(\Delta^2)$ leaves is at most $O(\log \Delta)$. Thus, the size of a path R from a leaf to the root is at most $O(\log \Delta)$. The vertex v only awakens in rounds corresponding to keys appearing along R, and thus v awakens in at most $O(\log \Delta)$ rounds. This provides us with the complexity of our algorithm in the following theorem.

▶ **Theorem 4.2.** *Any* **1-O-LOCAL** *problem can be solved in* $O(\log \Delta + \log^* n)$ *deterministic awake-complexity in the sleeping model.*

Note that each vertex is able to accumulate all information received from outgoing neighbors and pass it later to incoming neighbors, when these neighbors ask it for its output. Consequently, each vertex learns the information from all vertices that emanate from it in the orientation. Thus, each vertex is able to produce an output not only as a function of its outgoing-neighbors outputs, but as a function of all output of vertices that emanate from it. In other words, any **O-LOCAL** problem can be solved this way. Hence, we obtain the following corollary.

▶ **Corollary 4.3.** *Any* **O-LOCAL** *problem can be solved in* $O(\log \Delta + \log^* n)$ *deterministic awake-complexity in the sleeping model.*

In addition to the above results we also obtained further results. They are omitted from this version of the paper, due to space limitations. We refer the reader to the full version of this paper for a detailed description of these results [5].

5 Conclusion

In this work we investigated the strength of Distributed Layered Trees in the sleeping model. We showed that the computation of such trees is complete and thus any decidable problem can be solved within $O(\log n)$ awake complexity. This raises the question of finding non-trivial sub-classes of decidable problems which one can solve in a more efficient way than using a DLT. We address this question by defining the **O-LOCAL** class of problems and showing that it indeed can be solved more efficiently in the sleeping model. Since the $\mathcal{CONGEST}$ model is of great interest in the field of distributed networks, we investigated it as well, and obtained a class of problems that can be solved within logarithmic awake complexity by using only messages of logarithmic size. Another important aspect is the number of ordinary clock rounds of an algorithm with good awake complexity. While our simpler version of the algorithm has quadratic complexity of clock rounds, the more sophisticated variant gets closer to the optimal $\Theta(n)$ rounds. Overall, we showed the strength of the sleeping model and the possibility of a significant energy conservation for distributed networks.

References

1 Alkida Balliu, Fabian Kuhn, and Dennis Olivetti. Distributed edge coloring in time quasi-polylogarithmic in delta. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 289–298. ACM, 2020. doi:10.1145/3382734.3405710.

2 Leonid Barenboim, Shlomi Dolev, and Rafail Ostrovsky. Deterministic and energy-optimal wireless synchronization. *ACM Trans. Sens. Networks*, 11(1):13:1–13:25, 2014. `doi:10.1145/2629493`.

3 Leonid Barenboim and Michael Elkin. Deterministic distributed vertex coloring in polylogarithmic time. *J. ACM*, 58(5):23:1–23:25, 2011. `doi:10.1145/2027216.2027221`.

4 Leonid Barenboim, Michael Elkin, and Tzalik Maimon. Deterministic distributed (delta + o(delta))-edge-coloring, and vertex-coloring of graphs with bounded diversity. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 175–184. ACM, 2017. `doi:10.1145/3087801.3087812`.

5 Leonid Barenboim and Tzalik Maimon. Deterministic logarithmic completeness in the distributed sleeping model. *CoRR*, abs/2108.01963, 2021. `arXiv:2108.01963`.

6 Leonid Barenboim and Yaniv Tzur. Distributed symmetry-breaking with improved vertex-averaged complexity. In *Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019*, pages 31–40. ACM, 2019. `doi:10.1145/3288599.3288601`.

7 Milan Bradonjic, Eddie Kohler, and Rafail Ostrovsky. Near-optimal radio use for wireless network synchronization. *Theor. Comput. Sci.*, 453:14–28, 2012. `doi:10.1016/j.tcs.2011.09.026`.

8 Yi-Jun Chang, Varsha Dani, Thomas P. Hayes, Qizheng He, Wenzheng Li, and Seth Pettie. The energy complexity of broadcast. In Calvin Newport and Idit Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 95–104. ACM, 2018. `doi:10.1145/3212734.3212774`.

9 Soumyottam Chatterjee, Robert Gmyr, and Gopal Pandurangan. Sleeping is efficient: MIS in $O(1)$-rounds node-averaged awake complexity. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*, pages 99–108. ACM, 2020. `doi:10.1145/3382734.3405718`.

10 Jing Deng, Yunghsiang S. Han, Wendi Rabiner Heinzelman, and Pramod K. Varshney. Scheduling sleeping nodes in high density cluster-based sensor networks. *Mob. Networks Appl.*, 10(6):825–835, 2005. `doi:10.1007/s11036-005-4441-9`.

11 Cynthia Dwork, Joseph Y. Halpern, and Orli Waarts. Performing work efficiently in the presence of faults. *SIAM J. Comput.*, 27(5):1457–1491, 1998. `doi:10.1137/S0097539793255527`.

12 Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Proceedings IEEE INFOCOM 2001, The Conference on Computer Communications, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Twenty years into the communications odyssey, Anchorage, Alaska, USA, April 22-26, 2001*, pages 1548–1557. IEEE Comptuer Society, 2001. `doi:10.1109/INFCOM.2001.916651`.

13 Laurent Feuilloley. How long it takes for an ordinary node with an ordinary ID to output? In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, volume 10641 of *Lecture Notes in Computer Science*, pages 263–282. Springer, 2017. `doi:10.1007/978-3-319-72050-0_16`.

14 Laurent Feuilloley. How long it takes for an ordinary node with an ordinary id to output? *Theor. Comput. Sci.*, 811:42–55, 2020. `doi:10.1016/j.tcs.2019.01.023`.

15 Manuela Fischer. Improved deterministic distributed matching via rounding. *Distributed Comput.*, 33(3-4):279–291, 2020. `doi:10.1007/s00446-018-0344-4`.

16 Manuela Fischer, Mohsen Ghaffari, and Fabian Kuhn. Deterministic distributed edge-coloring via hypergraph maximal matching. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*, pages 180–191. IEEE Computer Society, 2017. `doi:10.1109/FOCS.2017.25`.

17 Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, 1987. `doi:10.1145/7531.7919`.

18 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983. doi:10.1145/357195.357200.

19 Leszek Gasieniec, Erez Kantor, Dariusz R. Kowalski, David Peleg, and Chang Su. Time efficient k-shot broadcasting in known topology radio networks. *Distributed Comput.*, 21(2):117–127, 2008. doi:10.1007/s00446-008-0058-0.

20 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 784–797. ACM, 2017. doi:10.1145/3055399.3055471.

21 Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1998, San Francisco, California, USA*, pages 219–225. ACM/SIAM, 1998. URL: http://dl.acm.org/citation.cfm?id=314613.314705.

22 Valerie King, Cynthia A. Phillips, Jared Saia, and Maxwell Young. Sleeping on the job: Energy-efficient and robust broadcast for radio networks. *Algorithmica*, 61(3):518–554, 2011. doi:10.1007/s00453-010-9422-0.

23 Fabian Kuhn. Faster deterministic distributed coloring through recursive list coloring. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1244–1259. SIAM, 2020. doi:10.1137/1.9781611975994.76.

24 Nathan Linial. Distributive graph algorithms-global solutions from local data. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 331–335. IEEE Computer Society, 1987. doi:10.1109/SFCS.1987.20.

25 Miao Peng, Yang Xiao, and Pu Patrick Wang. Error analysis and kernel density approach of scheduling sleeping nodes in cluster-based wireless sensor networks. *IEEE Trans. Veh. Technol.*, 58(9):5105–5114, 2009. doi:10.1109/TVT.2009.2027908.

26 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 350–363. ACM, 2020. doi:10.1145/3357713.3384298.

27 Jukka Suomela. Survey of local algorithms. *ACM Comput. Surv.*, 45(2):24:1–24:40, 2013. doi:10.1145/2431211.2431223.

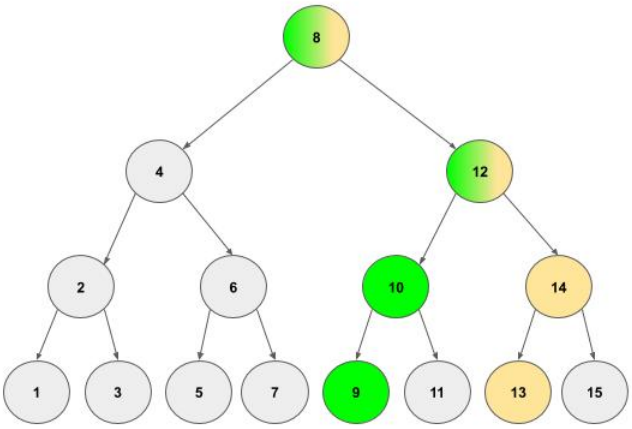
28 Chafiq Titouna, Abdelhak Mourad Guéroui, Makhlouf Aliouat, Ado Adamou Abba Ari, and Amine Adouane. Distributed fault-tolerant algorithm for wireless sensor networks. *Int. J. Commun. Networks Inf. Secur.*, 9(2), 2017. URL: http://www.ijcnis.org/index.php/ijcnis/article/view/1848.

29 Lijun Wang, Jia Yan, Tao Han, and Dexiang Deng. On connectivity and energy efficiency for sleeping-schedule-based wireless sensor networks. *Sensors*, 19(9):2126, 2019. doi:10.3390/s19092126.

30 Rong Zheng and Robin Kravets. On-demand power management for ad hoc networks. *Ad Hoc Networks*, 3(1):51–68, 2005. doi:10.1016/j.adhoc.2003.09.008.

A An Example for a Colored Binary Tree in the O-LOCAL Algorithm

An example of a tree in the internal memory of each processor, for $q = 8$. A pair of neighbors u, v (not depicted) are colored by 9 and 13, respectively. In green are the rounds in which vertices that correspond to color 9 are awake. In orange are the rounds in which the vertices that correspond to color 13 are awake. The lowest common ancestor of these two colors is 12. In this round both u and v awake, and v receives the decision of u. (Note that both u and v are also awake in round 8, but in this round u may have not reached a decision yet, since its color is $9 > 8$.)



B Proof for Lemma 4.1

Proof. We prove the lemma by induction on the colors of the orientation.

Base: For the left-most leaf in the tree, the mapping of the first color in the orientation maps to the first awakening round of the algorithm. Vertices with the first colors of the orientation have no outgoing edges and need not wait for decisions of any of their neighbors. As they wake at the first round they make a decision to be in the MIS and sleep again.

Step: Let v be a vertex which awakes in round r' and assume by induction that all neighbors with lower colors already made a decision to be in the MIS or not. Let $\hat{\Delta}$ be the number of neighbors of v with colors smaller than the color of v. Let $S = s_1, \ldots, s_{\hat{\Delta}}$ be the rounds mapped to each color of these neighbors. Then we have $r' > \{s_i \in S\}$. Thus, in the binary tree, for each $i \in [\hat{\Delta}]$, r' and s_i have a lowest common ancestor with ID t, such that $s_i \ < t < r'$. (See Figure 1.) This is because a lowest common ancestor of two leaves must have these leaves in distinct subtrees rooted in its children. Otherwise, if both leaves belong to the same subtree of a child of a common ancestor, it is not the lowest one.

Let u be a neighbor of v with a color corresponding to the mapping s_i. We note that s_i must be in the subtree rooted in the left child of the ancestor of ID t and r' is in the subtree rooted in the right-child of the ancestor t. Both u and v are awakened in round t according to our algorithm. At round t, since $s_i < t$, the vertex u already made a decision if it is in the MIS or not, by the induction hypothesis. Thus, u sends a message with its decision to v at round t. Since $r' > t$, at round t, v simply receives the messages and awaits round r' to make a decision. (During this waiting period, the vertex v may communicate with additional neighbors.) When round r' finally arrives, all neighbors with lower colors, those in S, have made decisions and sent their decision in the round corresponding to some common ancestor with v in the binary tree. Thus, v has learnt the decisions of neighbors with smaller colors than its own. Finally, v makes a decision in round r' according to all the decisions made by neighbors in S. This concludes the proof of the lemma. ◀

C Psuedo-Code for DLT Construction

Algorithm 1 Connection(G).

```
/******** First Connection Stage ***********/
for each tree $T \in G$ in parallel do
    Scan all edges $(u, w) \in G$, such that $u \in T$ and $w \notin T$. Search for an edge $e = (u, w)$, such that the first coordinate in $L(w)$ is the smallest, and this coordinate is smaller than the ID of $T$.
    if such an edge $e$ was found then
        Connect $T$ to the tree to which $w$ belongs, and make that tree a parent of $T$.
        The vertex $u$ becomes the new root of $T$. Set the label of each vertex $v \in T$ to be $\langle L(T), l \rangle$ where $l = dist_T(u, v)$.
    else
        Mark $T$ as a local minimum tree.
    end if
end for
/******** Second Connection Stage ***********/
for each local minimum tree $T'_i$ in parallel do
    Perform a convergecast in $T'_i$ to check if $T'_i$ became a parent of another tree.
    if $T'_i$ did not became a parent of another tree then
        Choose an edge $e'$ with one endpoint in $T'_i$, connecting to another connected component arbitrarily.
    end if
end for
Wake up all vertices in $G$ for one round and exchange knowledge about all edges $e'$ that cross between components $T'_i$, $T'_j$, $i \neq j$. This is achieved by sending component labels over the edges $e'$ in parallel within one round. In the end of this round, all vertices enter the sleep state.
for each connected component $C$ in parallel do
    Using convergecast collect information about all vertices of $C$ in the root of $C$.
    The root $r_C$ of $C$ executes a BFS internally and reassigns labels to each vertex in $v \in C$, such that the new label is $\langle L(C), l \rangle$, where $l = dist_C(u, v)$. In this reassignment, the root $r_c$ remains the same.
    Broadcast the result to all vertices in $C$.
end for
```

Wait-Free CAS-Based Algorithms: The Burden of the Past

Denis Bédin
Université de Nantes, France

François Lépine
Université de Nantes, France

Achour Mostéfaoui
LS2N, Université de Nantes, France

Damien Perez
Université de Nantes, France

Matthieu Perrin
LS2N, Université de Nantes, France

Abstract

Herlihy proved that CAS is universal in the classical computing system model composed of an a priori known number of processes. This means that CAS can implement, together with reads and writes, any object with a sequential specification. For this, he proposed the first universal construction capable of emulating any data structure. It has recently been proved that CAS is still universal in the infinite arrival computing model, a model where any number of processes can be created on the fly (e.g. multi-threaded systems). In this paper, we prove that CAS does not allow to implement wait-free and linearizable visible objects in the infinite model with a space complexity bounded by the number of active processes (i.e. ones that have operations in progress on this object). This paper also shows that this lower bound is tight, in the sense that this dependency can be made as low as desired (e.g. logarithmic) by proposing a wait-free and linearizable universal construction, using the compare-and-swap operation, whose space complexity in the number of ever issued operations is defined by a parameter that can be linked to any unbounded function.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Process synchronization; Computer systems organization → Multicore architectures; Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Compare-And-Swap, Concurrent Object, Infinite arrival model, Linearizability, Memory complexity, Multi-Threaded Systems, Shared-Memory, Universality, Wait-freedom

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.11

Funding This work was partially supported by the French ANR project 16-CE25-0005 O'Browser (`http://obrowser.univ-nantes.fr/`) devoted to the study of decentralized applications on Web browsers.

1 Introduction

Synchronization appeared with the concurrency brought by the first parallel programs in the early sixties. Concurrent accesses to shared data or any physical or logical resource by multiple processes can lead to inconsistencies. Dijkstra introduced the famous mutual-exclusion problem and proposed to solve it using locks [7]. Since then it is still one of the most popular mechanisms for inter-process synchronisation due to its supposed simplicity. The simplest way to implement mutual exclusion on uni-processor systems is by interruption disabling. Interestingly, it turns out that locks can also be implemented using read and write operations on shared variables [8]. However, these implementations have a space complexity linear with the number of processes [4]. This drawback has been overcome with the introduction of hardware special instructions like compare-and-set, test-and-set, fetch-and-add, etc. These instructions, referred to as read-modify-write instructions, aim to avoid certain interleavings in the execution of the processes by making it possible to read and update a memory location in one atomic operation. They represent, in some sense, a seed of atomicity. The compare-and-set instruction (CAS) is certainly one of the most popular (a.k.a. `compare_exchange_strong` in C++ and `compareAndSet` in Java). It is supported by most modern multiprocessor and multi-core architectures. Informally, the CAS operation has three arguments: the address of a memory location and two values. The memory location is set to the second value if, and only if, the first value is equal to the one stored by the memory location, and a Boolean result (success or failure) is returned to the calling process.

However, locks don't compose and do not tolerate process crashes. If a process holding a lock fails, the whole computation will stuck. Prohibiting the use of locks led to several progress conditions, among which wait-freedom [10] and lock-freedom [12][1]. While wait-freedom guarantees that every operation invoked by a non crashed process terminates after a finite time, lock-freedom guarantees that, if a computation runs for long enough, at least one process makes progress (this may lead some other processes to starve). Wait-freedom is thus stronger than lock-freedom: while lock-freedom is a system-wide progress condition, wait-freedom is a per-process progress condition.

Coordination between processes that access shared resources can be captured as concurrent data structures [2, 6, 11]. The design of the most popular concurrent data structures such as counters, queues, stacks, logs, etc. has been very active these last three decades. Unfortunately, not all data structures admit linearizable wait-free implementations in an asynchronous crash-prone concurrent system that only offers read and write basic operations on variables. This is due to the impossibility to solve the Consensus problem deterministically in this model [13]. The formulation of the Consensus problem is particularly simple. Each process proposes a value and all non-faulty processes decide on the same value among those which are proposed. In contrast, consensus was proved universal in [10]. Namely, any object having a sequential specification has a wait-free linearizable implementation using only read/write basic operations and some number of consensus objects. Moreover, some – but not all – special instructions, such as compare-and-set (CAS) or load-link/store-conditional (LL/SC), are universal as well. Therefore, while special hardware instructions provide efficiency in lock-based computing, they are necessary for lock-free and wait-free computing.

In order to prove the universality of consensus, Herlihy introduced the notion of universal construction. It is a generic algorithm that can emulate any object from its sequential specification. Since then, several universal constructions have been proposed for different special hardware instructions such as CAS and LL/SC [16]. Those are usually designed by first introducing a lock-free universal construction, which is then made wait-free with the use of helping: when a process invokes an operation, it first announces it in a dedicated single-writer variable, and then helps all other announced operations to terminate. Valency-based Helping has recently been proved to be unavoidable for CAS-based implementations of several data structures [5]. Hence, similarly to the space complexity drawbacks described above for the implementations of locks using only read and write operations on shared variables, many wait-free universal constructions have a space complexity linear in the number of potential participating processes. This inefficiency concerns both lock-free and wait-free implementations and is related to the use of historyless objects such as registers, LL/SC, CAS,

[1] In [12] lock-freedom is called non-blocking.

and TAS for example. It has been proved in [9] that the minimal space complexity of the implementations that use only historyless objects is linear with the number of participating processes.

In 2000, Merritt and Taubenfeld [14] introduced the infinite arrival model to deal with computing systems composed of an unbounded number of processes unlike the classical model composed of a fixed and a priori known number of processes. This model includes among others the multi-threaded model where any number of threads can be created and started at run-time and may leave or crash. So although the number of processes at each time instant is finite, it is not a priori known and there is no bound on the total number of threads that can participate in long-running executions. Recently, the universality of consensus and CAS has been extended to the infinite arrival model [15] by proposing a universal construction. In the proposed construction, helping is managed by an announcement data structure in which newly arrived processes can safely insert their operation. Unfortunately, terminated operations cannot be removed, resulting in an ever-growing data structure whose size depends on the total number of ever issued operations.

Problem Statement. This paper explores the performance aspect of the synchronization based on the CAS hardware special instruction. More precisely, we ask the following question: *Is it possible to design a wait-free universal construction whose space complexity only depends on the number of operations in progress?*.

Contribution 1: We prove that the answer is negative when only read, write and compare-and-set operations are available. This means that the space complexity depends on the total number of processes that ever issued operations, and that complex data structures must be maintained, and traversed, to implement helping mechanisms.

Contribution 2: Conversely, we show that our lower bound is tight, in the sense that this dependency can be made as low as desired (e.g. logarithmic), as long as it remains unbounded. We present a wait-free and linearizable universal construction, using the compare-and-set operation, whose space complexity in the number of ever issued operations is defined by a parameter that can be linked to any unbounded function. Obviously, this low spatial complexity is obtained to the detriment of the time complexity.

Let us note that lock-free linearizable implementations can trivially have a constant space complexity when no operation is in progress by simply having a CAS in a loop.

Organization of the paper. The remainder of this paper is organized as follows. In Section 2, we present the computing model that we consider and we define some notions that will be used afterwards. Then, Sections 3 and 4 respectively present the lower and upper bounds on the space complexity of wait-free and linearizable CAS-based algorithms. Finally, Section 5 concludes the paper.

2 Model

This paper considers the infinite arrival model [14] composed of a countable set Π of asynchronous sequential processes $p_0, p_1, \ldots$ that have access to a shared memory. The set Π is the set of potential processes that may join, get started and crash or leave during a given execution. At any time, the number of processes that have already joined the system is finite, but can be infinitely growing in long-running executions. Each process p_i has a unique identifier i that may appear in its code.

2.1 Communication between processes

Processes have access to local memory for local computations and have also access to a shared memory to communicate and synchronize. The shared memory is composed of an infinite number of unbounded locations, called registers [2]. Processes have access to a dynamic memory allocation mechanism accessible through the syntax **new** T, that instantiates an object of type T (T may be `Reg` to allocate a single register, as well as a record datatype or an array) and returns its reference, i.e. it allocates the memory locations needed to manage the object and initializes them by calling a constructor. Processes are not limited in the number of locations they can access, nor by the number of times they can use the allocation mechanism, during an execution. However, they can only access memory locations that either 1) have been allocated at the system set up, or 2) are returned by the allocation mechanism, or 3) are accessible by following references stored (as integer values) in some accessible memory location. In other words, when a process p_i allocates memory locations at runtime, they can initially only be accessed by p_i until it manages to share a reference pointing to these new memory locations. We say that a memory location is *reachable* when it can be accessed by a newly arrived process. When a memory location becomes inaccessible by any process in the system, it is automatically de-allocated by a garbage collector mechanism.

Processes can read the value of a shared register x by invoking x.`read`(), and can write a value v in x by invoking x.`write`(v). Moreover, as some objects cannot be implemented using only read/write operations, the system is enriched with the special atomic instruction `CAS`. Reads, writes and `CAS` are atomic in the sense that the different executions of the calls to these operations are totally ordered.

The compare-and-set instruction can be invoked on a register x with the expression x.`CAS`(*expect*, *update*), which returns a Boolean value. In the execution, the value stored in x is first compared to *expect*. If they are equal, then *update* is written in the register and **true** is returned. Otherwise, the state is left unchanged and **false** is returned.

2.2 Concurrent executions

An execution α is a (finite or infinite) sequence of steps, each taken by a process of Π. A step of a process corresponds to the execution of a read, a write, or `CAS`. Processes are asynchronous, in the sense that there is no constraint on which process takes each step: a process may take an unbounded number of consecutive steps, or wait an unbounded but finite number of other processes' steps between two of its own steps. This makes it possible to abstract from the difference in load of the different processors (cores) and from the fact that access to a processor is controlled by a scheduler. Moreover, it is possible that a process stops taking steps at some point in the execution, in which case we say this process has *crashed*, or even that a process takes no step during a whole execution (Π is only a set of potential participants). We say that a process p_i *arrives* in an execution at the time of its first step during this execution. Remark that, although the number of processes in an execution may be infinite in the infinite arrival model, the number of processes that have already arrived into the system at any step is finite.

[2] The assumption of an infinite memory, also made in the definition of Turing Machines, abstracts the fact that modern memories are large enough for all applications we consider in this paper and allows for simpler reasoning. The assumption of unbounded memory locations is then necessary to store references as memory addresses of an infinite memory are unbounded. The reader can check that we do not use these assumptions in any unpractical way.

A configuration C is composed of the local state of each process in Π and the value of each location in the shared memory. For a finite execution α, we denote by $C(\alpha)$ the configuration obtained at the end of α. An empty execution is denoted ε. An execution β is an extension of α if α is a prefix of β.

2.3 Implementation of shared objects

An implementation of a shared object is an algorithm divided into a set of sub-algorithms, one for the initialization (a.k.a. the constructor of the object), and one for each operation of the object, that produces wait-free and linearizable executions.

▶ **Definition 1** (Linearizability). *An execution α is* linearizable *if all operations have the same effect and return the same value as if they occurred instantly at some point of the timeline, called the* linearization point*, between their invocation and their response, possibly after removing some non-terminated operations.*

▶ **Definition 2** (Wait-freedom). *An execution α is* wait-free *if no operation takes an infinite number of steps in α.*

In this paper, we are interested in the space complexity of implementations. We distinguish the space complexity necessary to processes during the execution of their operations (e.g. their local memory and the memory locations that will be garbage-collected at the end of their execution), and the long-lasting space requirements of the data structures necessary to store the metadata of the algorithm, and that remains allocated even after all processes have terminated their operations. More precisely, we aim at minimizing the *quiescent complexity*, that measures the memory space required to store the state of a shared object when no process is executing an operation on it. This is to make sure we do not count the local storage of processes, which is not meaningful since the number of processes is unbounded.

▶ **Definition 3** (Quiescent complexity). *Let A be an algorithm. A finite execution α of A is said to be n-*quiescent *if exactly n operations of A were invoked, and all of them are completed, in $C(\alpha)$.*

The quiescent *complexity of A is the function $QC : \mathbb{N} \mapsto \mathbb{N} \cup \{\infty\}$, where $QC(n)$ is the maximal number of memory locations reachable in some configuration $C(\alpha)$ obtained at the end of any n-quiescent execution α, if this maximum exists, and ∞ otherwise.*

As explained in the Introduction, a universal construction is a generic algorithm, parametrized by the specification of a shared object, called a *state machine*, and that emulates a wait-free, linearizable shared version of the state machine. In this paper, the sequential specification of a state machine is defined as a transition system, whose initial state is the constant `initialState`, and whose transitions are defined by a function `execute` that takes as arguments a state of the object and an operation, and returns a pair formed by the resulting state and the return value obtained when the input operation is executed sequentially on the input state. For example, the fact that the `dequeue` operation on a non-empty queue deletes the first element from the queue and returns it is specified as $\texttt{execute}([x_0, x_1, \dots, x_n], \texttt{dequeue}) = \langle [x_1, \dots, x_n], x_0 \rangle$.

3 Lower Bound on Universal constructions using Compare-And-Swap

This section explores the limitations of wait-free linearizable universal constructions based on compare-and-set. More precisely, Theorem 11 proves that there is no such construction with constant quiescent complexity, i.e. any such construction must maintain a data structure that may grow over time. Let us first introduce the notions of mute process and visible object. We call *mute process* (Definition 4) a process that lost all its attempts at compare-and-set, and all the values it wrote in shared variables were overwritten before they could be read by another process. The class of *visible objects*, as defined in [9], is "a class that includes all objects that support some operation that must perform a visible write before it terminates. This class includes many useful objects (counters, stacks, queues, swap, fetch-and-add, and single-writer snapshot objects)". Intuitively, any update operation on any visible object (Lemma 5 considers a linearizable counter for this matter) must modify the global state of the system, in order to have an impact on the value returned by subsequent reads. For CAS-based algorithms, it implies that the presence of mute processes cannot be known by any other process, so they cannot complete their update operations on their own, nor can they be helped by others.

▶ **Definition 4** (mutism). *Let α be a finite execution, and let p be a process. We say that p is* mute *in α if there exists an execution α' such that p did not participate in α' and, for all processes $p' \neq p$, all shared variables and the local state of p' are the same in $C(\alpha)$ and $C(\alpha')$, i.e. $C(\alpha)$ and $C(\alpha')$ are indistinguishable*[3] *to p'.*

A finite execution α is said to be mute *if there exists a mute process p that terminated its execution in $C(\alpha)$.*

▶ **Lemma 5.** *Let A be a wait-free linearizable implementation of a counter (i.e. containing one operation, `increment`, that returns the number of previous invocations to `increment`). Then A does not have a mute execution.*

Proof. Let A be a wait-free linearizable implementation of a counter. Suppose (by contradiction) that A has a mute execution α, and let n be the number of processes that participate in α.

By definition, there exists a process p and an execution α' such that p terminated its execution in $C(\alpha)$, p did not participate in α' and, for all processes $p' \neq p$, $C(\alpha)$ and $C(\alpha')$ are indistinguishable to p'.

Let us consider the extension $\alpha\beta\gamma$ of α such that in β, all processes p' that took steps in α terminate their invocation, and in γ, some process q that did not participate in α joined and completed an invocation of `increment` in isolation, getting n as a result (β and γ exist because A is wait-free). As α and α' are indistinguishable to all processes p' and to q, $\alpha'\beta\gamma$ is also a valid execution, in which q also gets n as a result. However, only $n-1$ invocations of `increment` were started before q terminated, so A is not linearizable. ◀

In an algorithm A with a constant quiescent complexity, a bounded number of memory locations may remain reachable forever after a certain point in time (Definition 6 calls them static), and other memory locations may be allocated by some operations, and later be made unreachable by the same or another operation (Definition 6 calls them dynamic). Lemma 10 builds an execution in which some process p remains mute forever because, whenever p covers a static location x (i.e. p is about to write in x, see Definition 7), some other process also covers x and wins the competition, and whenever p covers a dynamic location, this location is made unreachable, so p_i's write remains unnoticed.

[3] This means that process p has no way to distinguish the two configurations. As said in [1] "Lack of knowledge about other components can formally be captured through the concept of indistinguishability, namely inability to tell apart different behaviors or states of the environment. Indistinguishability is therefore a consequence of the fact that computer systems are built of individual components, each with its own perspective of the system".

▶ **Definition 6** (static vs dynamic locations). *Let α be a finite execution, and x be a shared memory location that is reachable in $C(\alpha)$. We say that x is* dynamic *in α if there exists an extension $\alpha\beta$ of α such that 1) x is unreachable in $C(\alpha\beta)$, 2) no process that is mute in α takes steps in β, and 3) all processes are either mute in $C(\alpha)$ or have terminated their execution in $C(\alpha\beta)$. We say that x is* static *in α if it is not dynamic in α.*

▶ **Definition 7** (covering). *Let α be a finite execution, p a process and x a shared variable. We say that p* write-covers *(resp.* CAS-covers*) x in $C(\alpha)$ if the next step of p in $C(\alpha)$ is a write (resp an invocation of* $\mathtt{CAS}(e,u)$ *with $e \neq u$) on x. We say that p* covers *x in $C(\alpha)$ if p write-covers or CAS-covers x in $C(\alpha)$.*

In order to simplify the proofs, we only consider, in Lemma 10, algorithms that follow a normal form, defined in Definition 8. This assumption is done without loss of generality, since the proof of Lemma 9 discusses how to normalize any algorithm.

▶ **Definition 8** (Normal form). *An algorithm A is said to be in* normal form *if it satisfies the following properties.*

1. *There exists a location* last *such that the last step of any process is a write in* last*, that is never accessed otherwise.*
2. *Each time a process p invokes* $x.\mathtt{CAS}(e,u)$*, its previous step is a read of x that returned e.*
3. *All values written, or proposed as the second argument of compare-and-set, are different.*

▶ **Lemma 9.** *Any wait-free linearizable implementation A of a counter in the infinite arrival model, with a constant quiescent complexity and that only uses read, write and compare-and-set operations can be converted into a wait-free linearizable implementation A' of a counter in normal form with a constant quiescent complexity.*

Proof. We transform A into an algorithm A' in normal form as follows. First, we add an integer shared variable last initialized to 0 (if A already contains a variable named last, this variable is renamed in A'). We also add a concluding step in which all processes write their identifier in last.

Then, we replace all shared registers by a modified register whose type is defined by Algorithm 1, keeping the same invocations to read, write and compare-and-set. To comply with the third property of the definition of a normal form, Algorithm 1 associates a unique timestamp with each value proposed to write and `CAS` operations, consisting of a sequence number `time` and a process identifier `pid`. For that, each process locally numbers its different write and `CAS` operations using a local variable cl_i and, since the different processes have unique identifiers, no confusion can occur between the timestamps forged by different processes. During a read operation, the timestamp is removed and only the value relevant to the object is returned by the read. Hence, Algorithm 1 uses one shared register, storing values from a structured type containing three fields: a field value storing the value relevant to A, an integer field `time` and an integer field `pid`.

Remark that Algorithm 1 is itself a wait-free and linearizable implementation of a shared register (using each time the last operation on internal as linearization point), so the algorithm A' verifies all liveness and safety properties proved on A. In particular, A' is also a wait-free linearizable implementation of a counter. Moreover, A' also has a constant quiescent complexity since we only added one static memory location last, and Algorithm 1 multiplies the size of all used memory locations by a constant factor. ◀

▶ **Lemma 10.** *All wait-free algorithms in normal form with constant quiescent complexity have a mute execution.*

Algorithm 1 Normalisation of shared registers : code for p_i.

```
constructor ($initial$) is
    internal $\leftarrow$ new Reg({value $\leftarrow$ $initial$, time $\leftarrow 0$, pid $\leftarrow 0$});
operation read() is
    return internal.read().value;
operation write($v$) is
    $v' \leftarrow$ {value $\leftarrow v$, time $\leftarrow cl_i$, pid $\leftarrow i$};
    $cl_i \leftarrow cl_i + 1$;
    internal.write($v'$);
operation CAS($e, u$) is
    $e' \leftarrow$ internal.read();
    if $e'$.value $\neq e$ then return false;
    $u' \leftarrow$ {value $\leftarrow u$, time $\leftarrow cl_i$, pid $\leftarrow i$};
    $cl_i \leftarrow cl_i + 1$;
    return internal.CAS($e', u'$);
```

Proof. Let A be a wait-free algorithm in normal form, and let us suppose there is a tight bound k on the quiescent complexity of A. We prove, by induction on i, the following claim for all $i \in \{0, \ldots, k\}$.

▶ **Predicate 1** ($P_1(i)$)**.** *For all finite executions α, there exists an extension $\alpha\beta$ of α in which no process participates in both α and β, at least i static locations are covered by mute processes in $C(\alpha\beta)$ that didn't participate in α, and all non-mute processes that took part in β have terminated their execution.*

Proof. The empty execution works for $P_1(0)$, as it concerns no location and no process. Suppose we have proved $P_1(i)$ for some $i < k$. We now prove, by induction on j, the following claim for all $j \in \{0, \ldots, i+1\}$.

▶ **Predicate 2** ($P_2(i, j)$)**.** *For all finite executions α, there exists an extension $\alpha\beta$ of α in which in which no process participates in both α and β, and either (1) at least $i+1$ static locations are covered in $C(\alpha\beta)$ by mute processes that did not participate in α, or (2) at least j static locations are write-covered in $C(\alpha\beta)$ by mute processes that did not participate in α, and all non-mute processes that took part in β have terminated their execution.*

Proof. Predicate $P_2(i, 0)$ is implied by $P_1(i)$. Suppose we have proved $P_2(i, j)$ for some $j \leq i$. We suppose (by contradiction), that $P_2(i, j+1)$ does not hold.

Let $p \in \Pi$ be a process that did not take any step in α. We build, inductively, a sequence $(\alpha_m)_{m \in \mathbb{N}}$ of executions such that $\alpha_0 = \alpha$, for all $m \in \mathbb{N}$, p is mute in α_m, $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$ is an extension of α_m, and if $m > 0$, p takes at least one step in $\beta_m\gamma_m\delta_m$, a different set of processes participate in each extension $\beta_m\gamma_m\delta_m$, and all processes $q \neq p$ that take a step in $\beta_m\gamma_m\delta_m$ are terminated immediately after taking their step. Suppose we have built α_m for some $m \in \mathbb{N}$. In β_m, p takes steps in isolation until it covers some reachable location x. As A is wait-free, either such a situation is bound to happen or p is mute, which concludes the proof of $P_2(i, j)$. Three cases are to be distinguished.

- Suppose x is a dynamic location in $\alpha_m\beta_m$. There exists an extension $\alpha_m\beta_m\gamma_m$ of $\alpha_m\beta_m$ in which x is not reachable, and only mute processes or processes that have terminated their execution know the existence of x. Let $\alpha_m\beta_m\gamma_m\delta_m$ be the extension of $\alpha_m\beta_m\gamma_m$ in which p takes one step.

- Otherwise, x is a static location. Suppose p write-covers x in $C(\alpha_m\beta_m)$. Let $\alpha_m\beta_m\gamma_m$ be the extension of $\alpha_m\beta_m$ provided by Predicate $P_2(i,j)$. As we supposed that $P_2(i,j+1)$ does not hold, the only possibility is that at most j static locations are write-covered in $C(\alpha_m\beta_m\gamma_m)$, by mute processes that did not participate in $\alpha_m\beta_m$, including x, that is write-covered by some process q. In δ_m, p first takes one step, writing in x, and then q completes its execution, overwriting p's write. Therefore p is mute in $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$, and p took one step in δ_m.
- Otherwise, x is a static location and p's next step in $C(\alpha_m\beta_m)$ is $x.\texttt{CAS}(e,u)$, with $e \neq u$ since A is in normal form. Let $\alpha_m\beta_m\gamma_m$ be the extension of $\alpha_m\beta_m$ provided by Predicate $P_1(i)$. As we supposed that $P_2(i,j+1)$ does not hold, the only possibility is that at least i static locations are covered in $C(\alpha_m\beta_m\gamma_m)$, by mute processes that did not participate in $\alpha_m\beta_m$, including x, that is covered by some process q. If q writes-covers x, we build δ_m as previously. Otherwise, q's next step in $C(\alpha_m\beta_m\gamma_m)$ is $x.\texttt{CAS}(e',u')$, with $e' \neq u' \neq e$, by property (3) of the normal form, since A is in normal form.

  Let $\bar{x}$ be the value stored in x in Configuration $C(\alpha_m\beta_m\gamma_m)$. If $\bar{x} = e'$, in δ_m, q first takes one step, writing in u' in x, then p takes one step, that does not change the value of x and returns **false** ($u' \neq e$), and finally q terminates its execution. Therefore p is mute in $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$, and p took one step in δ_m.

  Otherwise, as A is in normal form, $\bar{x}$ was written in x after q read e' during γ_m, which occurred after p read e during β_m, so $e \neq \bar{x}$. In δ_m, then p takes one step, that does not change the value of x and returns **false**. Therefore p is mute in $\alpha_{m+1} = \alpha_m\beta_m\gamma_m\delta_m$, and p took one step in δ_m.

Finally, p takes an infinite number of steps in $\alpha\beta_1\gamma_1\delta_1\beta_2\gamma_2\delta_2\ldots$ without terminating, which contradicts the fact that A is wait-free. This terminates the proof of $P_2(i,j)$ for all $j \in \{0,\ldots,i+1\}$. ◀

Let us come back to the proof of Predicate $P_1(i+1)$. By $P_2(i,i+1)$, there exists an extension $\alpha\beta$ of α in which at least $i+1$ static locations are covered in $C(\alpha\beta)$, i.e. $P_1(i+1)$ is true. This terminates the proof of $P_1(i)$ for all $i \in \{0,k\}$. ◀

Finally, by invoking $P_1(k)$ twice, there exists an extension α of the empty execution ε in which k static locations are covered in $C(\alpha)$, and an extension $\alpha\beta$ of α in which k static locations are covered in $C(\alpha\beta)$ by processes that did not participate in α.

Let p be a process that did not participate in $\alpha\beta$. As all processes that participate in $\alpha\beta$ are either mute or have terminated their execution in $C(\alpha\beta)$, there exists an execution γ such that $C(\alpha\beta)$ and $C(\gamma)$ are indistinguishable to p, and no process is executing A in $C(\gamma)$. By definition of k, at most k locations are reachable in $C(\gamma)$, so at most k locations are reachable in $C(\alpha\beta)$ as well. By items items 2 and 3 of Definition 6, all k are static.

Therefore, all k locations are covered at least twice by mute processes in $C(\alpha\beta)$. In particular, there are two mute processes q and r that are about to write in last. Let us pose δ the sequence of steps in which q writes in last and then completes its invocation, and then p writes in last. Process q is mute in $\alpha\beta\delta$ and q terminates its execution, so $\alpha\beta\delta$ is a mute execution of A. ◀

▶ **Theorem 11.** *There is no wait-free linearizable implementation of a counter with a constant quiescent complexity in the infinite arrival model, that only uses read, write and compare-and-set operations.*

Proof. Suppose there is a wait-free linearizable implementation A of a counter with a constant quiescent complexity. In particular, by Lemma 9, we can suppose without loss of generality that it is in normal form. By Lemma 10, A has a mute execution, and by Lemma 5, A does not have a mute execution. This is a contradiction, so A does not exist. ◀

4 Upper Bound on Universal constructions using Compare-And-Swap

From Theorem 11, we can derive that the quiescent complexity of any wait-free linearizable universal construction is in $\omega(1)$. Differently, [3] presents such a construction with a quiescent complexity in $\mathcal{O}(n)$. The present section closes the gap thanks to Algorithm 2, a wait-free and linearizable universal construction that is parametrized by any unbounded and monotonically increasing function $f : \mathbb{N} \mapsto \mathbb{R}$ (e.g. log or $\log^\star$), and whose quiescent consistency is $QC(n) = \mathcal{O}(f(n))$. In the remainder of this section, let us fix an unbounded and monotonically increasing function f, and let us define its inverse f^{-1} as follows: for all $x \in \mathbb{N}$, $f^{-1}(x)$ is the smallest $y \in \mathbb{N}$ such that $f(y) \geq x$.

In Algorithm 2, a new operation is linearized each time a compare-and-set is won on a shared register linearization. In order to require the help from other processes, each operation starts by installing itself into a memory location that was at the head of a linked list announces when it started the algorithm. Once a process has failed too many times (depending on f) to install its operation, it changes the head of the linked list, which guarantees it not to lose again against any new operation.

Algorithm 2 maintains a data structure depicted on Figure 1, and composed of three kinds of nodes, described thereafter as structured data types.

- The first kind of nodes, called *operation node* and of type ONode, represents an ongoing operation. An operation node o is composed of three fields: o.oper is an operation of the state machine, o.result is a register storing either $\perp$ or a value that can be returned by o.oper, and o.done is a Boolean register. An operation node o is created when an operation o.oper is invoked by a process p_i on the state machine, initially with o.result $= \perp$ and o.done $=$ **false** (Line 8). After the operation has been linearized, o.done is set to **true** by some process p_j (possibly different to p_i) (Line 27), which serves as a signal to p_i that it can return o.result (Lines 16-17).
- The role of an *announce node* of type ANode is to expose a memory location in which a process can install an operation node, so that other processes can help completing the operation. An announce node a is either the empty node $\perp_a$ or a structure of two fields: a.next references another announce node and a.o is a register that references an operation node. In other words, an announce node is part of a linked list ending with $\perp_a$. We define the rank rank(a) of a node a as the length of the linked list, i.e. rank($\perp_a$) $= 0$ and rank(a) $=$ rank(a.next) $+ 1$ if $a \neq \perp_a$. The number of announce nodes accessible at the end of quiescent executions can only grow, which determines the quiescent complexity of Algorithm 2.
- Finally, a *linearization node* of type LNode represents a possible state of the state machine, as well as some information concerning the last operation leading to this state. A linearization node l is composed of three fields: l.state is a state of the state machine, l.result is a value returned by an operation of the state machine and s.o references an operation node. The sequence of states visited during an execution corresponds to a sequence of successful compare-and-set operations on linearization nodes.

Processes share two variables. The first one, announces, is a register that references announce nodes and is initialized to an announce node of rank one. The linked list of announce nodes accessible through announces provides a set of memory locations in which operation nodes can be placed to allow communication between processes that need helping and processes willing to help. The second variable, linearization, is a register that references a linearization node and is initialized to a new linearization node referencing the initial state of the state machine and a reference to a new dummy operation node. Later, linearization is composed of the current state of the state machine, as well as the operation node of the last linearized operation and its return value.

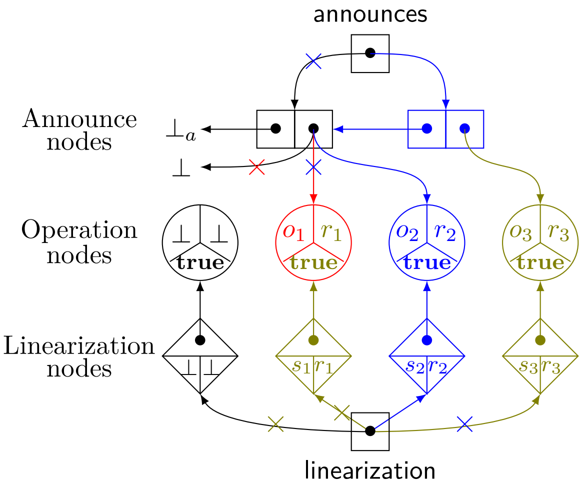


Figure 1 An execution of Algorithm 2, with $f(1) = 1$. The initial state is represented in black. Processes p_1 (in red), p_2 (in blue) and p_3 (in green) attempt to concurrently execute o_1, o_2 and o_3, respectively. Initially, p_1 and p_2 read the same announce node a, and p_1 wins the first compare-and-set, so p_2 creates a new announce node a' to prevent concurrency of newly arrived processes. Indeed, p_3 reads a' and writes its own operation node in it, then linearizes o_1 and o_3 and terminates. Finally, p_2 wins the compare-and-set on a and linearizes o_2.

When a process p_i needs to apply an operation op_i on the state machine, it invokes $\texttt{invoke}(op_i)$ on the universal construction. Process p_i first creates an operation node o_i containing its operation (Line 8), and then strives at installing o_i at the head a_i of the list of announce nodes referenced by announces, using compare-and-set (Line 18), after helping operation nodes already announced to be linearized by calling $\texttt{help}(a_i)$ (Line 14). If p_i fails to write o_i into a_i.o $f^{-1}(\texttt{rank}(a_i) + 1)$ times, it tries to insert a new announce node at the head of the announces list (Lines 11 and 13) to prevent newly arrived processes to compete on a_i.o, and ensure its own termination. Remark that p_i can only fail if some other process succeeded in inserting another announce node, providing the same benefits.

When p_i executes $\texttt{help}(a_i)$ to linearize the operation a_i.o.oper of the operation node a_i.o, it first helps recursively all announce nodes reachable from a_i (Line 21), and then tries to replace the linearization node in linearization using compare-and-set, until success (Lines 25 to 31). Remark that the new state of the state machine, as well as the value returned by an operation, are computed (Line 29), before the linearization node referencing the operation is created, and the return value is later reported on the operation node (Line 26), possibly by still a different process.

▶ **Lemma 12.** *No call to* $\texttt{help}(a_i)$ *in Algorithm 2 takes an infinite number of steps.*

Proof. Suppose, by contradiction, that some call to $\texttt{help}(a_i)$ by a process p_i takes an infinite number of steps. Without loss of generality, we can suppose that $r_i = \texttt{rank}(a_i)$ is minimal. Let o_i be the operation node read by p_i on Line 22. By Line 28, o_i.done is always **false**, so no process ever wins the compare-and-set on Line 31 with a linearization node referencing o_i (otherwise, the next process that writes linearization on Line 31 would previously have set o_i.done to **true** on Line 27), and $a_i.\mathsf{o} = o_i$ at all time after some point.

In only a finite number K of invocations of $\texttt{invoke}(op_j)$ by some process p_j, all done before the invocation of $\texttt{help}(a_i)$ by p_i, p_j reads an announce node a_j with $\texttt{rank}(a_j) < r_i$. All of them terminate because 1) by minimality of r_i, $\texttt{help}(a_k)$ terminates on Line 15 and 2)

Algorithm 2 Universal construction using compare-and-set.

```
1  constructor (initialState) is
2      $a_0 \leftarrow$ new ANode {next $\leftarrow \bot_a$, o $\leftarrow$ new Reg($\bot$)};
3      $o_0 \leftarrow$ new ONode {oper $\leftarrow \bot$, result $\leftarrow$ new Reg($\bot$), done $\leftarrow$ new Reg(true)};
4      $l_0 \leftarrow$ new LNode {state $\leftarrow$ initialState, result $\leftarrow \bot$, o $\leftarrow o_0$};
5      announces $\leftarrow$ new Reg($a_0$) ;
6      linearization $\leftarrow$ new Reg($l_0$) ;
7  operation invoke($op_i$) is
8      $o_i \leftarrow$ new ONode {oper $\leftarrow op_i$, result $\leftarrow$ new Reg($\bot$), done $\leftarrow$ new Reg(false)};
9      $a_i \leftarrow$ announces.read();
10     for $k \leftarrow 0, 1, 2, \ldots$ do
11         if $k = f^{-1}$(rank($a_i$) $+ 1$) then
12             $a'_i \leftarrow$ new ANode {next $\leftarrow a_i$, o $\leftarrow$ new Reg($\bot$)};
13             announces.CAS($a_i, a'_i$);
14         $o'_i \leftarrow a_i$.o.read();
15         help($a_i$);
16         if $o_i$.done.read() then
17             return $o_i$.result.read();
18         $a_i$.o.CAS($o'_i, o_i$);
19 function help($a_i$) is
20     if $a_i = \bot_a$ then return;
21     help($a_i$.next);
22     $o_i \leftarrow a_i$.o.read();
23     if $o_i = \bot$ then return;
24     while true do
25         $l_i \leftarrow$ linearization.read();
26         $l_i$.o.result.write($l_i$.result);
27         $l_i$.o.done.write(true);
28         if $o_i$.done.read() then return;
29         $\langle s_i, r_i \rangle \leftarrow$ execute($l_i$.state, $o_i$.oper);
30         $l'_i \leftarrow$ new LNode {state $\leftarrow s_i$, result $\leftarrow r_i$, o $\leftarrow o_i$};
31         linearization.CAS($l_i, l'_i$);
```

whenever a process wins a compare-and-set on Line 18, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 15) and then terminates on Line 17, so p_j can only be prevented to terminate K times. After that, a_j.o is never writen, and a_j.o.done remains **true** forever. In particular, no further invocation of help(a_j) executes Line 31, as they terminate on Line 28.

After that point, all new invocations of help(a_j) by some process p_j on Line 15 are such that rank(a_j) $\geq r_i$. Thanks to Line 21, and by what preceeds, p_j's first execution of Line 31 is during its recursive call help(a_i), in which p_j reads a_i.o $= o_i$ on Line 22. As all executions of Line 31 try to write $l_j = \langle s, r, o_i \rangle$ for some s and r, only one of them succeeds. After that point, some process (possibly p_i) reads l_j on Line 25 and writes **true** in o_i.done on Line 27, which is a contradiction. ◀

▶ **Lemma 13** (Wait-freedom). *Algorithm 2 is wait-free.*

Proof. Let us consider an invocation of $\texttt{invoke}(op_i)$ by a process p_i. By Lemma 12, the function `help` terminates, so all iterations of the loop by p_i terminate as well. Let $K = f^{-1}(\texttt{rank}(a_i) + 1)$. As f is unbounded, K is well defined.

If p_i iterates less than K times, then it terminates its execution. Otherwise, it executes Line 13 when $k = K$, and whether the compare-and-set is successful or not, $\mathsf{announces} \neq a_i$ after that. All processes that arrive later read a different value on Line 9, so only a finite number of processes compete with p_i on Line 18. Each time one of them succeeds, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 15) and then terminates on Line 17, so an operation can only prevent p_i to win its compare-and-set once. Therefore, p_i eventually terminates its execution. ◀

▶ **Lemma 14** (Linearizability). *All executions admitted by Algorithm 2 are linearizable.*

Proof. Let α be an execution admissible by Algorithm 2.

Let us first remark that, for any operation $\texttt{invoke}(op_i)$ invoked by process p_i, at most one linearization node l_i such that $l_i.\mathsf{o.oper} = op_i$ is such that an invocation of $\mathsf{linearization.CAS}(l', l_i)$ returns **true** on Line 31. Indeed, first remark that all linearization nodes written in $\mathsf{linearization}$ are unique, because they are created on the same line as they are written, and that only one operation node o_i, built on Line 8 by p_i, is such that $o_i.\mathsf{oper} = op_i$. Suppose (by contradiction) that two linearization nodes l_j and l_k, with $l_i.\mathsf{o} = l'_i.\mathsf{o} = o_i$, were successfully written in $\mathsf{linearization}$ by p_j and p_k respectively. Let us consider, without loss of generality, the first two such linearization nodes, and let us consider the linearization node l_m that overwrote l_i i.e. such that the invocation $\mathsf{linearization.CAS}(l_i, l_m)$ by some process p_m returned **true**. Process p_j read l in $\mathsf{linearization}$ on Line 25 before **false** in $o_i.\mathsf{done}$ on Line 28, before p_m wrote **true** in $o_i.\mathsf{done}$ on Line 27, before p_m invoked $\mathsf{linearization.CAS}(l_i, l_m)$ on Line 31. Therefore, l is at least as old as l_i. It is impossible that $l = l_i$ because it would mean $p_j = p_m$ would have executed Line 28 before Line 27, and it is impossible that l is older than l_i because it would have been overwritten by l_i or before.

Let us define the linearization point of any operation $\texttt{invoke}(op_i)$ as, if it exists, the unique successful invocation of $\mathsf{linearization.CAS}(l', l_i)$ such that $l_i.\mathsf{o.oper} = op_i$.

We now prove that any operation $\texttt{invoke}(op_i)$ done by a terminating process p_i has a linearization point, between its invocation and termination point. As p_i terminated, it read **true** in $o_i.\mathsf{done}$ on Line 16, so some process p_j wrote **true** in $l_i.\mathsf{o.done} = o_i.\mathsf{done}$ on Line 28, after having read l_i on Line 25, which can only happen after some process p_k wrote l_i on Line 31. This is a linearization point for $\texttt{invoke}(op_i)$. As we have seen, the linearization point happened before the p_i's termination. It also happened after p_i's invocation, as o_i can only be created by p_i on Line 8.

Finally, let us remark that, thanks to Line 29, the states and result values reached in a sequential execution E defined by the linearization order are the same as the ones written in the linearization nodes on Line 31. If Process p_i returns r_i at the end of the execution of $\texttt{invoke}(op_i)$, it read it in $o_i.\mathsf{result}.\texttt{read}()$ on Line 17, after reading **true** in $o_i.\mathsf{done}$ on Line 16, which can only happen if some process wrote **true** in $o_i.\mathsf{done}$ on Line 27 after writing $l_i.\mathsf{result}$ in $o_i.\mathsf{result}$ on Line 26, with $l_i.\mathsf{o} = o_i$ and l_i read in $\mathsf{linearization}$ on Line 25. Therefore, the p_i returns the same value as in H.

In conclusion, α is linearizable. ◀

▶ **Lemma 15** (Complexity). *The quiescent complexity of Algorithm 2 is $QC(n) = \mathcal{O}(f(n))$.*

Proof. Let α be a finite execution of Algorithm 2 such that n invocations of invoke(op_i) happened in α and all of them are completed in $C(\alpha)$.

Let r be the rank of the announcement node referenced by announces in $C(\alpha)$, and let us suppose that $r \geq 2$. Let us consider last time announces was updated in α, on Line 13, by a process p_i. Remark that whenever a process wins a compare-and-set on Line 18, it helps its own operation or a more recent one to terminate in its next iteration of the loop (Line 15) and then terminates on Line 17, so an operation can only prevent p_i to win its compare-and-set once. Therefore, $n \leq k = f^{-1}(r - 1 + 1) = f^{-1}(r)$. By definition of f^{-1}, we have $r \leq f(n)$.

One linearization node and at most $f(n)$ announce nodes, referencing at most $f(n)$ operation nodes, are reachable in $C(\alpha)$. Therefore, at most $O(f(n))$ shared memory locations dedicated to Algorithm 2 are reachable. ◀

5 Conclusion

This paper investigated the performance of concurrent data structure implementations (counters, queues, stacks, journals, etc.) in the infinite arrival model where the universal compare-and-set hardware instruction is available. It proves that the space complexity of a universal construction cannot be constant in the number of operations ever issued, although it can be super-constant.

This separation result may seem weak to separate only between constant and super-constant space, however, note that a low space complexity is obtained to the detriment of time complexity. This is captured by the function f. This function relates space complexity to the worst-case step/time complexity (f^{-1}); there is a kind of trade-off. This function can be seen as a continuum between wait-freedom and lock-freedom. While wait-freedom offers a finite time complexity and an ever-increasing space complexity, lock-freedom offers a constant quiescent space complexity and an infinite worst case time complexity (in a real setting and in the average, lock-free implementations are time efficient). The faster f grows, the closer we get to wait-freedom, and conversely, the slower the closer we get to lock-freedom. When parameterized with a slowly growing function, the proposed data structure can be as efficient as a lock-free data structure while benefiting from wait-freedom (the guarantee of a finite step complexity). An interesting open question, therefore, is whether other universal special hardware instructions can avoid this complexity issue.

References

1 Hagit Attiya and Sergio Rajsbaum. Indistinguishability. *Commun. ACM*, 63(5):90–99, 2020.

2 Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

3 Grégoire Bonin, Achour Mostéfaoui, and Matthieu Perrin. Wait-free universality of consensus in the infinite arrival model. In *33rd International Symposium on Distributed Computing, DISC, Hungary*, volume 146 of *LIPIcs*, pages 38:1–38:3, 2019.

4 James E. Burns, Paul Jackson, Nancy A. Lynch, Michael J. Fischer, and Gary L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *J. ACM*, 29(1):183–205, 1982.

5 Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 241–250, 2015.

6 David Dice, Danny Hendler, and Ilya Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, volume 8097 of *Lecture Notes in Computer Science*, pages 595–606. Springer, 2013.

7 Edsger Dijkstra. Over de sequentialiteit van procesbeschrijvingen (on the nature of sequential processes). *EW Dijkstra Archive (EWD-35), Center for American History, University of Texas at Austin (Translation by Martien van der Burgt and Heather Lawrence)*, 1962.

8 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

9 Faith E. Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In Soma Chaudhuri and Shay Kutten, editors, *Proc. of the 23rd Symposium on Principles of Distributed Computing, PODC 2004, Canada*, pages 80–87. ACM, 2004.

10 Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

11 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

12 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

13 Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Parallel and Distributed Computing*, 4(4):163–183, 1987.

14 Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proc. of International Symposium on Distributed Computing*, pages 164–178. Springer, 2000.

15 Matthieu Perrin, Achour Mostéfaoui, and Grégoire Bonin. Extending the wait-free hierarchy to multi-threaded systems. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 21–30, 2020.

16 Michel Raynal. Distributed universal constructions: a guided tour. *Bulletin of the EATCS*, 121, 2017.

Space and Time Bounded Multiversion Garbage Collection

Naama Ben-David
VMware Research, Palo Alto, CA, USA

Guy E. Blelloch
Carnegie Mellon University, Pittsburgh, PA, USA

Panagiota Fatourou
FORTH ICS and University of Crete, Heraklion, Greece

Eric Ruppert
York University, Toronto, Canada

Yihan Sun
University of California, Riverside, CA, USA

Yuanhao Wei
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We present a general technique for garbage collecting old versions for multiversion concurrency control that simultaneously achieves good time and space complexity. Our technique takes only $O(1)$ time on average to reclaim each version and maintains only a constant factor more versions than needed (plus an additive term). It is designed for multiversion schemes using version lists, which are the most common.

Our approach uses two components that are of independent interest. First, we define a novel *range-tracking data structure* which stores a set of old versions and efficiently finds those that are no longer needed. We provide a wait-free implementation in which all operations take amortized constant time. Second, we represent version lists using a new lock-free doubly-linked list algorithm that supports efficient (amortized constant time) removals given a pointer to any node in the list. These two components naturally fit together to solve the multiversion garbage collection problem–the range-tracker identifies which versions to remove and our list algorithm can then be used to remove them from their version lists. We apply our garbage collection technique to generate end-to-end time and space bounds for the multiversioning system of Wei et al. (PPoPP 2021).

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases Lock-free, data structures, memory management, snapshot, version lists

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.12

Related Version *Full Version*: `https://arxiv.org/abs/2108.02775`

Funding *Guy E. Blelloch and Yuanhao Wei*: NSF CCF-1901381, CCF-1910030, and CCF-1919223.
Eric Ruppert: NSERC Discovery Grant. Yihan Sun: NSF grant CCF-2103483.
Panagiota Fatourou: EU Horizon 2020, Marie Sklodowska-Curie GA No 101031688.

Acknowledgements We thank the anonymous referees for their helpful comments and suggestions.

1 Introduction

Supporting multiple "historical" versions of data, often called multiversioning or multiversion concurrency control, is a powerful technique widely used in database systems [42, 10, 38, 32, 36, 51], transactional memory [40, 22, 39, 31, 29], and shared data structures [7, 21, 35, 49].

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 12; pp. 12:1–12:20
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This approach allows complex queries (read-only transactions) to proceed concurrently with updates while still appearing atomic because they get data views that are consistent with a single point in time. If implemented carefully, queries do not interfere with one another or with updates. The most common approach for multiversioning uses version lists [42] (also called version chains): the system maintains a global timestamp that increases over time, and each object maintains a history of its updates as a list of value-timestamp pairs, each corresponding to a value written and an update time. Each node in the list has an associated interval of time from that node's timestamp until the next (later) node's timestamp. A query can first read a timestamp value t and then, for each object it wishes to read, traverse the object's version list to find the version whose interval contains t.

Memory usage is a key concern for multiversioning, since multiple versions can consume huge amounts of memory. Thus, most previous work on multiversioning discusses how to reclaim the memory of old versions. We refer to this as the multiversion garbage collection (MVGC) problem. A widely-used approach is to keep track of the earliest active query and reclaim the memory of any versions overwritten before the start of this query [22, 36, 30, 35, 49]. However, a query that runs for a long time, either because it is complicated or because it has been delayed, will force the system to retain many unneeded intermediate versions between the oldest required version and the current one. This has been observed to be a major bottleneck for database systems with Hybrid Transaction and Analytical Processing (HTAP) workloads [14] (i.e., many small updates concurrent with some large analytical queries). To address this problem in the context of software transactional memory, Lu and Scott [33] proposed a non-blocking algorithm that can reclaim intermediate versions. Blocking techniques were later proposed by the database community [14, 32]. However, these techniques add significant time overhead in worst-case executions.

We present a wait-free MVGC scheme that achieves good time and space bounds, using $O(1)$ time[1] on average per allocated version and maintaining only a constant factor more versions than needed (plus an additive term). The scheme is very flexible and it can be used in a variety of multiversioning implementations. It uses a three-step approach that involves 1) identifying versions that can be reclaimed, including intermediate versions, 2) unlinking them from the version lists, and 3) reclaiming their memory. To implement these three steps efficiently, we develop two general components – a *range-tracking* data structure and a *version-list* data structure – that could be of independent interest beyond MVGC.

The range-tracking data structure is used to identify version list nodes that are no longer needed. It supports an `announce` operation that is used by a query to acquire the current timestamp t as well as protect any versions that were current at t from being reclaimed. A corresponding `unannounce` is used to indicate when the query is finished. The data structure also supports a `deprecate` operation that is given a version and its time interval, and indicates that the version is no longer the most recent – i.e., is safe to reclaim once its interval no longer includes any announced timestamp. When a value is updated with a new version, the previous version is deprecated. A call to `deprecate` also returns a list of versions that had previously been `deprecated` and are no longer cover any announced timestamp – i.e., are now safe to reclaim. We provide a novel implementation of the range-tracking data structure for which the amortized number of steps per operation is $O(1)$. We also bound the number of versions on which `deprecate` has been called, but have not yet been returned. If H is the maximum, over all configurations, of the number of needed deprecated versions, then the number of deprecated versions that have not yet been returned is at most

[1] For time/space complexity, we count both local and shared memory operations/objects.

$2H + O(P^2 \log P)$, where P is the number of processes. To achieve these time and space bounds, we borrow some ideas from real-time garbage collection [6, 11], and add several new ideas such as batching and using a shared queue.

The second main component of our scheme is a wait-free version-list data structure that supports efficient (amortized constant time) removals of nodes from anywhere in the list. When the `deprecate` operation identifies an unneeded version, we must splice it out of its version list, without knowing its current predecessor in the list, so we need a doubly-linked version list. Our doubly-linked list implementation has certain restrictions that are naturally satisfied when maintaining version lists, for example nodes may be appended only at one end. The challenge is in achieving constant amortized time per remove, and bounded space. Previously known concurrent doubly-linked lists [47, 43] do not meet these requirements, requiring at least $\Omega(P)$ amortized time per remove. We first describe the implementation of our version list assuming a garbage collector and then we show how to manually reclaim removed nodes while maintaining our desired overall time and space bounds.

To delete elements from the list efficiently, we leverage some recent ideas from randomized parallel list contraction [12], which asynchronously removes elements from a list. To avoid concurrently splicing out adjacent elements in the list, which can cause problems, the approach defines an implicit binary tree so that the list is an in-order traversal of the tree. Only nodes corresponding to leaves of the tree, which cannot be adjacent in the list, may be spliced out. Directly applying this technique, however, is not efficient in our setting. To reduce space overhead, we had to develop intricate helping mechanisms for splicing out internal nodes rather than just leaves. To achieve wait-freedom, we had to skew the implicit tree so that it is right-heavy. The final algorithm ensures that at most $2(L - R) + O(P \log L_{max})$ nodes remain reachable in an execution with L appends and R removes across an arbitrary number of version lists, and at most L_{max} appends on a single version list. This means the version lists store at most a constant factor more than the $L - R$ required nodes plus an additive term shared across all the version lists. Combining this with the bounds from the range tracker, our MVGC scheme ensures that at most $O(V + H + P^2 \log P + P \log L_{max})$ versions are reachable from the V version lists. This includes the current version for each list, H needed versions, plus additive terms from the range tracking and list building blocks.

After a node has been spliced out of the doubly-linked list, its memory must be reclaimed. This step may be handled automatically by the garbage collector in languages such as Java, but in non-garbage-collected languages, additional mechanisms are needed to safely reclaim memory. The difficulty in this step is that while a node is being spliced out, other processes traversing the list might be visiting that node. We use a reference counting reclamation scheme and this requires modifying our doubly-linked list algorithm slightly to maintain the desired space bounds. We apply an existing concurrent reference counting implementation [2] that employs a local hash table per process which causes the time bounds of our reclamation to become amortized $O(1)$ *in expectation.* It also requires an additional fetch-and-add instruction, whereas the rest of our algorithms require only read and CAS.

We apply our MVGC scheme to a specific multiversioning scheme [49] to generate end-to-end bounds for a full multiversioning system. This multiversioning scheme takes a given CAS-based concurrent data structure and transforms it to support complex queries (e.g., range queries) by replacing each CAS object with one that maintains a version list. Overall, we ensure that the memory usage of the multiversion data structure is within a constant factor of the needed space, plus $O(P^2 \log P + P^2 \log L_{max})$. In terms of time complexity, our garbage collection scheme takes only $O(1)$ time on average for each allocated version.

Detailed proofs of correctness and of our complexity bounds appear in the full version [8].

2 Related Work

Garbage Collection. One of the simplest, oldest techniques for garbage collection is reference counting (RC) [16, 17, 28]. In its basic form, RC attaches to each object a counter of the number of references to it. An object is reclaimed when its counter reaches zero. Some variants of RC are wait-free [2, 46]. In Section 6, we apply the RC scheme of [2] to manage version list nodes as it adds only constant time overhead (in expectation) and it is the only concurrent RC scheme that maintains our desired time bounds.

Epoch-based reclamation (EBR) [23, 15] employs a counter that is incremented periodically and is used to divide the execution into epochs. Processes read and announce the counter value at the beginning of an operation. An object can be reclaimed only if it was retired in an epoch preceding the oldest announced. EBR is often the preferred choice in practice, as it is simple and exhibits good performance. However, a slow or crashed process with timestamp t can prevent the reclamation of *all* retired objects with timestamps larger than t. EBR, or variants, are used in a variety of MVGC schemes [22, 36, 49] to identify versions that are older than any query. An advantage of these schemes is that identified versions can be immediately reclaimed without first being unlinked from the version lists because the section of the version list they belong to is old enough to never be traversed. However, they inherit the same problem as EBR and are not able to reclaim intermediate versions between the oldest needed version and the current version when a long-running query holds on to an old epoch. This can be serious for multiversioned systems since EBR works best when operations are short, but a key motivation for multiversioning is to support lengthy queries.

Hazard pointers (HP) [28, 34] can be used to track which objects are currently being accessed by each process and are therefore more precise. Combinations of HP and EBR have been proposed (e.g. [41, 50]) with the goal of preserving the practical efficiency of EBR while lowering its memory usage. However, unlike EBR, none of these techniques directly solve the MVGC problem. Other memory reclamation schemes have been studied that require hardware support [1, 18] or rely on the signaling mechanism of the operating system [15, 45]. Hyaline [37] implements a similar interface to EBR and can be used for MVGC, but like EBR, it cannot reclaim intermediate versions.

We are aware of three multiversioning systems based on version lists that reclaim intermediate versions: GMV [33], HANA [32] and Steam [14]. To determine which versions are safe to reclaim, all three systems merge the current version list for an object with the list of active timestamps to check for overlap. The three schemes differ based on when they decide to perform this merging step and how they remove and reclaim version list nodes. In GMV, when an update operation sees that memory usage has passed a certain threshold, it iterates through all the version lists to reclaim versions. Before reclaiming a version, it has to help other processes traverse the version list to ensure traversals remain wait-free. HANA uses a background thread to identify and reclaim obsolete versions while Steam scans the entire version list whenever a new version is added to it. In HANA and Steam, nodes are removed by locking the entire version list, whereas in GMV, nodes are removed in a lock-free manner by first logically marking a node for deletion, as in Harris's linked list [26]. If a `remove` operation in GMV experiences contention (i.e., fails a CAS), it restarts from the head of the version list. None of these three techniques ensure constant-time removal from a version list. Both Steam and GMV ensure $O(PM)$ space where M is the amount of space required in an equivalent sequential execution. In comparison, we use a constant factor more than the required space plus an additive term of $O(P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of versions added to a single version list. This can be significantly less than $O(PM)$ in many workloads.

Lock-Free Data Structures and Query Support. We use doubly-linked lists to store old versions. Singly-linked lists had lock-free implementations as early as 1995 [48]. Several implementations of doubly-linked lists were developed later from multi-word CAS instructions [5, 24], which are not widely available in hardware but can be simulated in software [27, 25]. Sundell and Tsigas [47] gave the first implementation from single-word CAS, although it lacks a full proof of correctness. Shafiei [43] gave an implementation with a proof of correctness and amortized analysis. Existing doubly-linked lists are not efficient enough for our application, so we give a new implementation with better time bounds.

Fatourou, Papavasileiou and Ruppert [21] used multiversioning to add range queries to a search tree [19]. Wei et al. [49] generalized this approach (and made it more efficient) to support wait-free queries on a large class of lock-free data structures. Nelson, Hassan and Palmieri [35] sketched a similar scheme, but it is not non-blocking. In Appendix A, we apply our garbage collection scheme to the multiversion system of [49].

3 Preliminaries

We use a standard model with asynchronous, crash-prone processes that access shared memory using CAS, read and write instructions. For our implementations of data structures, we bound the number of steps needed to perform operations, and the number of shared objects that are allocated but not yet reclaimed.

We also use *destination objects* [13], which are single-writer objects that store a value and support `swcopy` operations in addition to standard reads and writes. A `swcopy(ptr)` atomically reads the value pointed to by `ptr`, and copies the value into the destination object. Only the owner of a destination object can perform `swcopy` and `write`; any process may `read` it. Destination objects can be implemented from CAS so that all three operations take $O(1)$ steps [13]. They are used to implement our range-tracking objects in Section 4.

Pseudocode Conventions. We use syntax similar to C++. The type `T*` is a pointer to an object of type `T`. `List<T>` is a `List` of objects of type `T`. If `x` stores a pointer to an object, then `x->f` is that object's member `f`. If `y` stores an object, `y.f` is that object's member `f`.

4 Identifying Which Nodes to Disconnect from the Version List

We present the *range-tracking* object, which we use to identify version nodes that are safe to disconnect from version lists because they are no longer needed. To answer a query, a slow process may have to traverse an entire version list when searching for a very old version. However, we need only maintain list nodes that are the potential target nodes of such queries. The rest may be spliced out of the list to improve space usage and traversal times.

We assign to each version node `X` an interval that represents the period of time when `X` was the current version. When the next version `Y` is appended to the version list, `X` ceases to be the current version and becomes a potential candidate for removal from the version list (if no query needs it). Thus, the left endpoint of `X`'s interval is the timestamp assigned to `X` by the multiversioning system, and the right endpoint is the timestamp assigned to `Y`.

We assume that a query starts by announcing a timestamp t, and then proceeds to access, for each relevant object o, its corresponding version at time t, by finding the first node in the version list with timestamp at most t (starting from the most recent version). Therefore, an announcement of t means it is unsafe to disconnect any nodes whose intervals contain t.

As many previous multiversioning systems [22, 32, 35, 36, 49] align with the general scheme discussed above, we define the range-tracking object to abstract the problem of identifying versions that are not needed. We believe this abstraction is of general interest.

▶ **Definition 1** (Range-Tracking Object). *A range-tracking object maintains a multiset A of integers, and a set O of triples of the form* `(o,low,high)` *where* `o` *is an object of some type T and* `low` $\leq$ `high` *are integers. Elements of A are called* active announcements*. If* `(o,low,high)` $\in O$ *then* `o` *is a* deprecated object *with associated half-open interval* [`low`, `high`)*. The range-tracking object supports the following operations.*

- `announce(int* ptr)` *atomically reads the integer pointed to by* `ptr`*, adds the value read to A, and returns the value read.*
- `unannounce(int i)` *removes one copy of* `i` *from A, rendering the announcement inactive.*
- `deprecate(T* o, int low, int high)`*, where* `low` $\leq$ `high`*, adds the triple* `(o,low,high)` *to O and returns a set S, which contains the deprecated objects of a set $O' \subseteq O$ such that for any $o \in O'$, the interval of o does not intersect A, and removes O' from O.*

The specification of Definition 1 should be paired with a progress property that rules out the trivial implementation in which `deprecate` always returns an empty set. We do this by bounding the number of deprecated objects that have not been returned by `deprecate`.

▶ **Assumption 2.** To implement the range-tracking object, we assume the following.

1. A process's calls to `deprecate` have non-decreasing values of parameter `high`.
2. If, in some configuration G, there is a pending `announce` whose argument is a pointer to an integer variable `x`, then the value of `x` at G is greater than or equal to the `high` argument of every `deprecate` that has been invoked before G.
3. For every process p, the sequence of invocations to `announce` and `unannounce` performed by p should have the following properties: a) it should start with `announce`; b) it should alternate between invocations of `announce` and invocations of `unannounce`; c) each `unannounce` should have as its argument the integer returned by the preceding `announce`.
4. Objects passed as the first parameter to `deprecate` operations are distinct.

In the context we are working on, we have a non-decreasing integer variable that works as a global timestamp, and is passed as the argument to every `announce` operation. Moreover, the `high` value passed to each `deprecate` operation is a value that has been read from this variable. This ensures that parts 1 and 2 of Assumption 2 are satisfied. The other parts of the assumption are also satisfied quite naturally for our use of the range-tracking object, and we believe that the assumption is reasonably general. Under this assumption, we present and analyze a linearizable implementation of the range-tracking object in Section 4.1.

4.1 A Linearizable Implementation of the Range-Tracking Object

Our implementation, RangeTracker, is shown in Figure 1. Assumption 2.3 means that each process can have at most one active announcement at a time. So, RangeTracker maintains a shared array `Ann` of length P to store active announcements. `Ann[p]` is a destination object (defined in Section 3) that is owned by process `p`. Initially, `Ann[p]` stores a special value $\perp$. To announce a value, a process `p` calls `swcopy` (line 28) to copy the current timestamp into `Ann[p]` and returns the announced value (line 29). To deactivate an active announcement, p writes $\perp$ into `Ann[p]` (line 31). Under Assumption 2.3, the argument to `unannounce` must match the argument of the process's previous `announce`, so we suppress `unannounce`'s argument in our code. An `announce` or `unannounce` performs O(1) steps.

```
 1 class Range { T* t, int low, int high; };
 2 class RangeTracker {
 3   // global variables
 4   Destination Ann[P];
 5   Queue<List<Range>> Q; //initially empty
 6   // thread local variables
 7   List<Range> LDPool; // initially empty
 8   Array<int> sortAnnouncements() {
 9     List<int> result;
10     for(int i = 0; i < P; i++) {
11       int num = Ann[i].read();
12       if(num != ⊥) result.append(num); }
13     return sort(toArray(result)); }

15   List<T*>, List<Range> intersect(
16       List<Range> MQ, Array<int> ar) {
17     Range r; int i = 0;
18     List<T*> Redundant;
19     List<Range> Needed;
20     for(r in MQ) {
21       while(i < ar.size() &&
22             ar[i] < r.high) i++;
23       if(i == 0 || ar[i-1] < r.low)
24         Redundant.append(r.t);
25       else Needed.append(r); }
26     return <Redundant, Needed>; }
27 int Announce(int* ptr) {
28   Ann[p].swcopy(ptr);
29   return Ann[p].read(); }

31 void unannounce() { Ann[p].write(⊥); }

33 List<T*> deprecate(T* o, int low, int high) {
34   List<T*> Redundant;
35   List<Range> Needed, Needed1, Needed2;
36   // local lists are initially empty
37   LDPool.append(Range(o, low, high));
38   if(LDPool.size() == B) {
39     List<Range> MQ = merge(Q.deq(),Q.deq());
40     Array<int> ar = sortAnnouncements();
41     Redundant, Needed = intersect(MQ, ar);
42     if(Needed.size() > 2*B) {
43       Needed1, Needed2 = split(Needed);
44       Q.enq(Needed1);
45       Q.enq(Needed2); }
46     else if(Needed.size() > B) {
47       Q.enq(Needed); }
48     else {
49       LDPool = merge(LDPool,Needed); }
50     Q.enq(LDPool);
51     LDPool = empty list; }
52   return Redundant; } };
```

Figure 1 Code for process p for our linearizable implementation of a range-tracking object.

A Range object (line 1) stores the triple (o,low,high) for a deprecated object o. It is created (at line 37) during a deprecate of o. RangeTracker maintains the deprecated objects as *pools* of Range objects. Each pool is sorted by its elements' high values. Each process maintains a local pool of deprecated objects, called LDPool. To deprecate an object, a process simply appends its Range to the process's local LDPool (line 37). Assumption 2.1 implies that objects are appended to LDPool in non-decreasing order of their high values.

We wish to ensure that most deprecated objects are eventually returned by a deprecate operation so that they can be freed. If a process p with a large LDPool ceases to take steps, it can cause all of those objects to remain unreturned. Thus, when the size of p's LDPool hits a threshold B, they are flushed to a shared queue, Q, so that other processes can also return them. The elements of Q are pools that each contain B to $2B$ deprecated objects. For the sake of our analysis, we choose $B = P \log P$. When a flush is triggered, p dequeues two pools from Q and processes them as a batch to identify the deprecated objects whose intervals do not intersect with the values in Ann, and return them. The rest of the dequeued objects, together with those in LDPool, are stored back into Q. We call these actions (lines 38–51), the *flush phase* of deprecate. A deprecate without a flush phase returns an empty set.

During a flush phase, a process p dequeues two pools from Q and merges them (line 39) into a new pool, MQ. Next, p makes a local copy of Ann and sorts it (line 40). It then uses the intersect function (line 41) to partition MQ into two sorted lists: Redundant contains objects whose intervals do not intersect the local copy of Ann, and Needed contains the rest. Intuitively, a deprecated object in MQ is put in Redundant if the low value of its interval is larger than the announcement value immediately before its high value. Finally, p enqueues the Needed pool with its LDPool into Q (lines 44–47 and line 50). To ensure that the size of each pool in Q is between B and $2B$, the Needed pool is split into two halves if it is too large (line 43), or is merged with LDPool if it is too small (line 49). A flush phase is performed once every $P \log P$ calls to deprecate, and the phase executes $O(P \log P)$ steps. Therefore, the amortized number of steps for deprecate is $O(1)$.

The implementation of the concurrent queue `Q` should ensure that an element can be enqueued or dequeued in $O(P \log P)$ steps. The concurrent queue presented in [20] has step complexity $O(P)$ and thus ensures these bounds. To maintain our space bounds, the queue nodes must be reclaimed. This can be achieved if we apply hazard-pointers on top of the implementation in [20]. If `Q` is empty, then `Q.deq()` returns an empty list.

We sketch the proofs of the following three theorems. For detailed proofs, see [8].

▶ **Theorem 3.** *If Assumption 2 holds, then* RangeTracker *is a linearizable implementation of a range-tracking object.*

The linearization points used in the proof of Theorem 3 are defined as follows. An `announce` is linearized at its `swcopy` on line 28. An `unannounce` is linearized at its `write` on line 31. A `deprecate` is linearized at line 50 if it executes that line, or at line 37 otherwise.

The most interesting part of the proof concerns a `deprecate` operation I with a flush phase. I dequeues two pools from `Q` as `MQ` and decides which objects in `MQ` to return based on the local copy of `Ann` array. To show linearizability, we must also show that intervals of the objects returned by I do not intersect the `Ann` array at the linearization point of I. Because of Assumption 2.2, values written into `Ann` after the pools are dequeued cannot be contained in the intervals in `MQ`. Thus, if an object's interval does not contain the value I read from `Ann[i]`, it will not contain the value in `Ann[i]` at I's linearization point.

▶ **Theorem 4.** *In the worst case,* `announce` *and* `unannounce` *take $O(1)$ steps, while* `deprecate` *takes $O(P \log P)$ steps. The amortized number of steps performed by each operation is $O(1)$.*

Let H be the maximum, over all configurations in the execution, of the number of *needed* deprecated objects, i.e., those whose intervals contain an active announcement.

▶ **Theorem 5.** *At any configuration, the number of deprecated objects that have not yet been returned by any instance of* `deprecate` *is at most $2H + 25P^2 \log P$.*

At any time, each process holds at most $P \log P$ deprecated objects in `LDPool` and at most $4P \log P$ that have been dequeued from `Q` as part of a flush phase. We prove by induction that the number of deprecated objects in `Q` at a configuration G is at most $2H + O(P^2 \log P)$. Let G' be the latest configuration before G such that all pools in `Q` at G' are dequeued between G' and G. Among the dequeued pools, only the objects that were needed at G' are re-enqueued into `Q`, and there are at most H such objects. Since we dequeue two pools (containing at least B elements each) each time we enqueue B *new* objects between G' and G, this implies that the number of such new objects is at most half the number of objects in `Q` at G' (plus $O(P^2 \log P)$ objects from flushes already in progress at G'). Assuming the bound on the size of `Q` holds at G', this allows us to prove the bound at G.

The constant multiplier of H in Theorem 5 can be made arbitrarily close to 1 by dequeuing and processing k pools of `Q` in each flush phase instead of two. The resulting space bound would be $\frac{k}{k-1} \cdot H + \frac{(2k+1)(3k-1)}{k-1} \cdot P^2 \log P$. This would, of course, increase the constant factor in the amortized number of steps performed by `deprecate` (Theorem 4).

5 Maintaining Version Lists

We use a restricted version of a doubly-linked list to maintain each version list so that we can more easily remove nodes from the list when they are no longer needed. We assume each node has a timestamp field. The list is initially empty and provides the following operations.

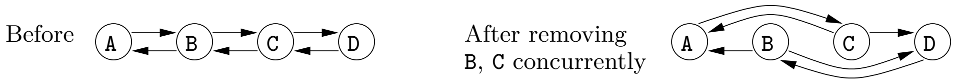


Figure 2 An example of incorrect removals.

- `tryAppend(Node* old, Node* new)`: Adds `new` to the head of the list and returns true if the current head is `old`. Otherwise returns false. Assumes `new` is not null.
- `getHead()`: Returns a pointer to the `Node` at the head of the list (or `null` if list is empty).
- `find(Node* start, int ts)`: Returns a pointer to the first `Node`, starting from `start` and moving away from the head of the list, whose timestamp is at most `ts` (or `null` if no such node exists).
- `remove(Node* n)`: Given a previously appended `Node`, removes it from the list.

To obtain an efficient implementation, we assume several preconditions, summarized in Assumption 6 (and stated more formally in the full version [8]). A version should be removed from the object's version list only if it is not current: either it has been superseded by another version (6.1) or, if it is the very last version, the entire list is no longer needed (6.2). Likewise, a version should not be removed if a `find` is looking for it (6.3), which can be guaranteed using our range-tracking object. We allow flexibility in the way timestamps are assigned to versions. For example, a timestamp can be assigned to a version after appending it to the list. However, some assumptions on the behaviour of timestamps are needed to ensure that responses to `find` operations are properly defined (6.4, 6.5).

▶ **Assumption 6.**

1. Each node (except the very last node) is removed only after the next node is appended.
2. No `tryAppend`, `getHead` or `find` is called after a `remove` on the very last node.
3. After `remove(X)` is invoked, no pending or future `find` operation should be seeking a timestamp in the interval between `X`'s timestamp and its successor's.
4. Before trying to append a node after a node `B` or using `B` as the starting point for a `find`, `B` has been the head of the list and its timestamp has been set. A node's timestamp does not change after it is set. Timestamps assigned to nodes are non-decreasing.
5. If a `find(X,t)` is invoked, any node appended after `X` has a higher timestamp than `t`.
6. Processes never attempt to append the same node to a list twice, or to remove it twice.

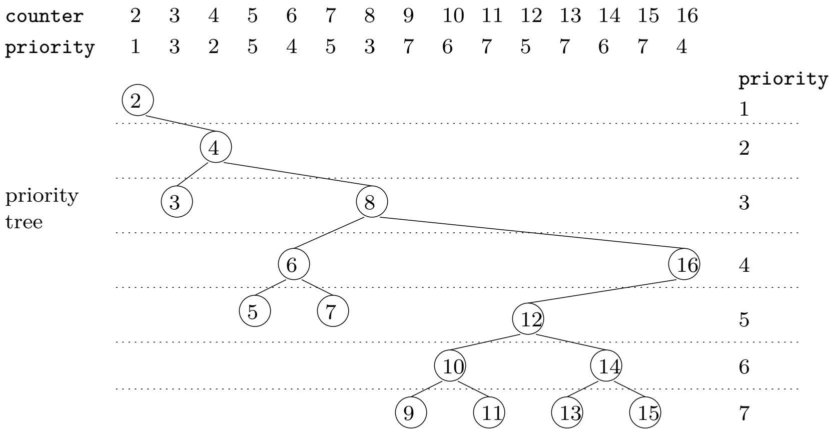
5.1 Version List Implementation

Pseudocode for our list implementation is in Figure 4. A `remove(X)` operation first marks the node `X` to be deleted by setting a `status` field of `X` to `marked`. We refer to the subsequent physical removal of `X` as *splicing* `X` out of the list.

Splicing a node `B` from a doubly-linked list requires finding its left and right neighbours, `A` and `C`, and then updating the pointers in `A` and `C` to point to each other. Figure 2 illustrates the problem that could arise if adjacent nodes `B` and `C` are spliced out concurrently. The structure of the doubly-linked list becomes corrupted: `C` is still reachable when traversing the list towards the left, and `B` is still reachable when traversing towards the right. The challenge of designing our list implementation is to coordinate splices to avoid this situation.

We begin with an idea that has been used for parallel list contraction [44]. We assign each node a priority value and splice a node out only if its priority is greater than both of its neighbours' priorities. This ensures that two adjacent nodes cannot be spliced concurrently.

Conceptually, we can define a *priority tree* corresponding to a list of nodes with priorities as follows. Choose the node with minimum priority as the root. Then, recursively define the left and right subtrees of the root by applying the same procedure to the sublists to the



counter	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
priority	1	3	2	5	4	5	3	7	6	7	5	7	6	7	4

Figure 3 A list and its priority tree.

left and right of the root node. The original list is an in-order traversal of the priority tree. See Figure 3 for an example. We describe below how we choose priorities to ensure that (1) there is always a unique minimum in a sublist corresponding to a subtree (to be chosen as the subtree's root), and (2) if L nodes are appended to the list, the height of the priority tree is $O(\log L)$. We emphasize that the priority tree is not actually represented in memory; it is simply an aid to understanding the design of our implementation.

The requirement that a node is spliced out of the list only if its priority is greater than its neighbours corresponds to requiring that we splice only nodes whose descendants in the priority tree have all already been spliced out of the list. To remove a node that still has unspliced descendants, we simply mark it as logically deleted and leave it in the list. If `X`'s descendants have all been spliced out, then `X`'s parent `Y` in the priority tree is the neighbour of `X` in the list with the larger priority. An operation that splices `X` from the list then attempts to *help* splice `X`'s parent `Y` (if `Y` is marked for deletion and `Y` is larger than its two neighbours), and this process continues up the tree. Conceptually, this means that if a node `Z` is marked but not spliced, the last descendant of `Z` to be spliced is also responsible for splicing `Z`.

In this scheme, an unmarked node can block its ancestors in the priority tree from being spliced out of the list. For example, in Figure 3, if the nodes with counter values 10 to 16 are all marked for deletion, nodes 11, 13 and 15 could be spliced out immediately. After 13 and 15 are spliced, node 14 could be too. The unmarked node 9 prevents the remaining nodes 10, 12 and 16 from being spliced, since each has a neighbour with higher priority. Thus, an unmarked node could prevent up to $\Theta(\log L)$ marked nodes from being spliced out of the list.

Improving this space overhead factor to $O(1)$ requires an additional, novel mechanism. If an attempt to remove node `B` observes that `B`'s left neighbour `A` is unmarked and `B`'s priority is greater than `B`'s right neighbour `C`'s priority, we allow `B` to be spliced out of the list using a special-purpose routine called `spliceUnmarkedLeft`, even if `A`'s priority is greater than `B`'s. In the example of the previous paragraph, this would allow node 10 to be spliced out after 11. Then, node 12 can be spliced out after 10 and 14, again using `spliceUnmarkedLeft`, and finally node 16 can be spliced out. A symmetric routine `spliceUnmarkedRight` applies if `C` is unmarked and `B`'s priority is greater than `A`'s. This additional mechanism of splicing out nodes when one neighbour is unmarked allows us to splice out all nodes in a string of consecutive marked nodes, except possibly one of them, which might remain in the list if both its neighbours are unmarked and have higher priority. However, during the `spliceUnmarkedLeft` routine that is splicing out `B`, `A` could become marked. If `A`'s priority is greater than its two neighbours' priorities, there could then be simultaneous splices of A and B. To avoid this, instead of splicing out B directly, the spliceUnmarkedLeft installs a pointer to a *Descriptor* object into node A, which describes the splice of B. If A becomes marked, the information in the Descriptor is used to *help* complete the splice of B before A itself is spliced. Symmetrically, a spliceUnmarkedRight of B installs a Descriptor in C.

Multiple processes may attempt to splice the same node B, either because of the helping coordinated by Descriptor objects or because the process that spliced B's last descendant in the priority tree will also try to splice B itself. To avoid unnecessary work, processes use a CAS to change the status of B from marked to finalized. Only the process that succeeds in this CAS has the responsibility to recursively splice B's ancestors. (In the case of the spliceUnmarkedLeft and spliceUnmarkedRight routines, only the process that successfully installs the Descriptor recurses.) If one process responsible for removing a node (and its ancestors) stalls, it could leave $O(\log L)$ marked nodes in the list; this is the source of an *additive* $P \log L$ term in the bound we prove on the number of unnecessary nodes in the list.

We now look at the code in more detail. Each node X in the doubly-linked list has right and left pointers that point toward the list's head and away from it, respectively. X also has a status field that is initially unmarked and leftDesc and rightDesc fields to hold pointers to Descriptors for splices happening to the left and to the right of X, respectively. X's counter field is filled in when X is appended to the right end of the list with a value that is one greater than the preceding node. To ensure that the height of the priority tree is $O(\log L)$, we use the counter value c to define the priority of X as $p(c)$, where $p(c)$ is either k if c is of the form 2^k, or $2k+1-$(number of consecutive 0's at the right end of the binary representation of c), if $2^k < c < 2^{k+1}$. The resulting priority tree has a sequence of nodes with priorities $1, 2, 3, \ldots$ along the rightmost path in the tree, where the left subtree of the ith node along this rightmost path is a complete binary tree of height $i-1$, as illustrated in Figure 3. (Trees of this shape have been used to describe search trees [9] and in concurrent data structures [3, 4].) This assignment of priorities ensures that between any two nodes with the same priority, there is another node with lower priority. Moreover, the depth of a node with counter value c is $O(\log L)$. This construction also ensures that remove operations are wait-free, since the priority of a node is a bound on the number of recursive calls that a remove performs.

A Descriptor of a splice of node B out from between A and C is an object that stores pointers to the three nodes A, B and C. After B is marked, we set its Descriptor pointers to a special Descriptor frozen to indicate that no further updates should occur on them.

To append a new node C after the head node B, the tryAppend(B,C) operation simply fills in the fields of C, and then attempts to swing the Head pointer to C at line 36. B's right pointer is then updated at line 37. If the tryAppend stalls before executing line 37, any attempt to append another node after C will first help complete the append of C (line 32). The boolean value returned by tryAppend indicates whether the append was successful.

A remove(B) first sets B's status to marked at line 44. It then stores the frozen Descriptor in both B->leftDesc and B->rightDesc. The first attempt to store frozen in one of these fields may fail, but we prove that the second will succeed because of some handshaking, described below. B is *frozen* once frozen is stored in both of its Descriptor fields. Finally, remove(B) calls removeRec(B) to attempt the real work of splicing B.

The removeRec(B) routine manages the recursive splicing of nodes. It first calls splice, spliceUnmarkedLeft or spliceUnmarkedRight, as appropriate, to splice B. If the splice of B was successful, it then recurses (if needed) on the neighbour of B with the larger priority.

The actual updates to pointers are done inside the splice(A,B,C) routine, which is called after reading A in B->left and C in B->right. The routine first tests that A->right = B at line 96. This could fail for two reasons: B has already been spliced out, so there is no

```
class Node {
  Node *left, *right; // initially null
  enum status {unmarked,marked,finalized};
      // initially unmarked
  int counter; // used to define priority
  int priority; // defines implicit tree
  int ts; // timestamp
  Descriptor *leftDesc, *rightDesc;
      // initially null
};

class Descriptor { Node *A, *B, *C; };
Descriptor* frozen = new Descriptor();

class VersionList {
  Node* Head;
  // public member functions:
  Node* getHead() {return Head;}

  Node* find(Node* start, int ts) {
    VNode* cur = start;
    while(cur != null && cur->ts > ts)
      cur = cur->left;
    return cur; }

  bool tryAppend(Node* B, Node* C) {
    // B can be null iff C is the initial node
    if(B != null) {
      C->counter = B->counter+1;
      Node* A = B->left;
      // Help tryAppend(A, B)
      if(A != null) CAS(&(A->right), null, B);
    } else C->counter = 2;
    C->priority = p(C->counter);
    C->left = B;
    if(CAS(&Head, B, C)) {
      if(B != null) CAS(&(B->right), null, C);
      return true;
    } else return false; }

  // public static functions:
  void remove(Node* B) {
    // B cannot be null
    B->status = marked;
    for F in [leftDesc, rightDesc] {
      repeat twice {
        Descriptor* desc = B->F;
        help(desc);
        CAS(&(B->F), desc, frozen); } }
    removeRec(B); }

  // private helper functions:
  bool validAndFrozen(Node* D) {
    // rightDesc is frozen second
    return D != null && D->rightDesc == frozen; }

  void help(Descriptor* desc) {
    if(desc != null && desc != frozen)
      splice(desc->A, desc->B, desc->C); }

  int p(int c) {
    k = floor(log2(c));
    if(c == 2^k) return k;
    else return 2k + 1 - lowestSetBit(c); }
```

```
// private helper functions continued:
void removeRec(Node* B) {
  // B cannot be null
  Node* A = B->left;
  Node* C = B->right;
  if(B->status == finalized) return;
  int a, b, c;
  if(A != null) a = A->priority;
  else a = 0;
  if(C != null) c = C->priority;
  else c = 0;
  b = B->priority;
  if(a < b > c) {
    if(splice(A, B, C)) {
      if(validAndFrozen(A)) {
        if(validAndFrozen(C) && c > a) removeRec(C);
        else removeRec(A); }
      else if(validAndFrozen(C)) {
        if(validAndFrozen(A) && a > c) removeRec(A);
        else removeRec(C); } } }
  else if(a > b > c) {
    if(spliceUnmarkedLeft(A, B, C) &&
    validAndFrozen(C)) {
      removeRec(C); } }
  else if(a < b < c) {
    if(spliceUnmarkedRight(A, B, C) &&
    validAndFrozen(A)) {
      removeRec(A); } } } }

bool splice(Node* A, Node* B, Node* C) {
  // B cannot be null
  if(A != null && A->right != B) return false;
  bool result = CAS(&(B->status), marked, finalized);
  if(C != null) CAS(&(C->left), B, A);
  if(A != null) CAS(&(A->right), B, C);
  return result; }

bool spliceUnmarkedLeft(Node* A, Node* B, Node* C) {
  // A, B cannot be null
  Descriptor* oldDesc = A->rightDesc;
  if(A->status != unmarked) return false;
  help(oldDesc);
  if(A->right != B) return false;
  Descriptor* newDesc = new Descriptor(A, B, C);
  if(CAS(&(A->rightDesc), oldDesc, newDesc)) {
    // oldDesc != frozen
    help(newDesc);
    return true;
  } else return false; }

bool spliceUnmarkedRight(Node* A, Node* B, Node* C) {
  // B, C cannot be null
  Descriptor* oldDesc = C->leftDesc;
  if(C->status != unmarked) return false;
  help(oldDesc);
  if(C->left != B || (A != null && A->right != B))
    return false;
  Descriptor* newDesc = new Descriptor(A, B, C);
  if(CAS(&(C->leftDesc), oldDesc, newDesc)) {
    // oldDesc != frozen
    help(newDesc);
    return true;
  } else return false; } };
```

Figure 4 Linearizable implementation of our doubly-linked list.

need to proceed, or there is a `splice(A,D,B)` that has been partially completed; `B->left` has been updated to `A`, but `A->right` has not yet been updated to `B`. In the latter case, the `remove` that is splicing out `D` will also splice `B` after `D`, so again there is no need to proceed with the splice of `B`. If `A->right` = `B`, `B`'s `status` is updated to `finalized` at line 97, and the pointers in `C` and `A` are updated to splice `B` out of the list at line 98 and 99.

The `spliceUnmarkedLeft(A,B,C)` handles the splicing of a node `B` when `B`'s left neighbour `A` has higher priority but is unmarked, and `B`'s right neighbour `C` has lower priority. The operation attempts to CAS a Descriptor of the splice into `A->rightDesc` at line 109. If there was already an old Descriptor there, it is first helped to complete at line 106. If the new Descriptor is successfully installed, the `help` routine is called at line 111, which in turn calls `splice` to complete the splicing out of `B`. The `spliceUnmarkedLeft` operation can fail in several ways. First, it can observe that `A` has become marked, in which case `A` should be spliced out before `B` since `A` has higher priority. (This test is also a kind of handshaking: once a node is marked, at most one more Descriptor can be installed in it, and this ensures that one of the two attempts to install `frozen` in a node's Descriptor field during the `remove` routine succeeds.) Second, it can observe at line 107 that `A->right` ≠ `B`. As described above for the `splice` routine, it is safe to abort the splice in this case. Finally, the CAS at line 109 can fail, either because `A->rightDesc` has been changed to `frozen` (indicating that `A` should be spliced before `B`) or another process has already stored a new Descriptor in `A->rightDesc` (indicating either that `B` has already been spliced or will be by another process).

The `spliceUnmarkedRight` routine is symmetric to `spliceUnmarkedLeft`, aside from a slight difference in line 120 because `splice` changes the `left` pointer before the `right` pointer. The return values of `splice`, `spliceUnmarkedLeft` and `spliceUnmarkedRight` say whether the calling process should continue recursing up the priority tree to splice out more nodes.

5.2 Properties of the Implementation

Detailed proofs of the following results appear in the full version [8]. We sketch them here.

▶ **Theorem 7.** *Under Assumption 6, the implementation in Figure 4 is linearizable.*

Since the implementation is fairly complex, the correctness proof is necessarily quite intricate. We say that `X` $<_c$ `Y` if node `X` is appended to the list before node `Y`. We prove that `left` and `right` pointers in the list always respect this ordering. Removing a node has several key steps: marking it (line 44), freezing it (second iteration of line 49), finalizing it (successful CAS at line 97) and then making it unreachable (successful CAS at line 99). We prove several lemmas showing that these steps take place in an orderly way. We also show that the steps make progress. Finally, we show that the coordination between `remove` operations guarantees that the structure of the list remains a doubly-linked list in which nodes are ordered by $<_c$, except for a temporary situation while a node is being spliced out, during which its left neighbour may still point to it after its right neighbour's pointer has been updated to skip past it. To facilitate the inductive proof of this invariant, it is wrapped up with several others, including an assertion that overlapping calls to `splice` of the form `splice(W,X,Y)` and `splice(X,Y,Z)` never occur. The invariant also asserts that unmarked nodes remain in the doubly-linked list; no `left` or `right` pointer can jump past a node that has not been finalized. Together with Assumption 6.3, this ensures a `find` cannot miss the node that it is supposed to return, regardless of how `find` and `remove` operations are linearized. We linearize `getHead` and `tryAppend` when they access the `Head` pointer.

▶ **Theorem 8.** *The number of steps a* `remove(X)` *operation performs is* $O($`X->priority`$)$ *and the* `remove` *operation is therefore wait-free.*

Proof. Aside from the call to `removeRec(X)`, `remove(X)` performs $O(1)$ steps. Aside from doing at most one recursive call to `removeRec`, a `removeRec` operation performs $O(1)$ steps. Each time `removeRec` is called recursively, the node on which it is called has a smaller priority. Since priorities are non-negative integers, the claim follows. ◀

▶ **Theorem 9.** *The* `tryAppend` *and* `getHead` *operations take $O(1)$ steps. The amortized number of steps for* `remove` *is $O(1)$.*

Consider an execution with R `remove` operations. Using the argument for Theorem 8, it suffices to bound the number of calls to `removeRec`. There are at most R calls to `removeRec` directly from `remove`. For each of the R nodes `X` that are removed, we show that at most one call to `removeRec(X)` succeeds either in finalizing `X` or installing a Descriptor to remove `X`, and only this `removeRec(X)` can call `removeRec` recursively.

We say a node is *lr-reachable* if it is reachable from the head of the list by following `left` or `right` pointers. A node is *lr-unreachable* if it is not lr-reachable.

▶ **Theorem 10.** *At the end of any execution by P processes that contains L successful* `tryAppend` *operations and R* `remove` *operations on a set of version lists, and a maximum of L_{max} successful* `tryAppend`*s on a single version list, the total number of lr-reachable nodes across all the version lists in the set is at most $2(L - R) + O(P \log L_{max})$.*

Theorem 10 considers a set of version lists to indicate that the $O(P \log L_{max})$ additive space overhead is shared across all the version lists in the system. A node `X` is *removable* if `remove(X)` has been invoked. We must show at most $(L - R) + O(P \log L_{max})$ removable nodes are still lr-reachable. We count the number of nodes that are in each of the various phases (freezing, finalizing, making unreachable) of the removal. There are at most P removable nodes that are not yet frozen, since each has a pending `remove` operation on it. There are at most P finalized nodes that are still lr-reachable, since each has a pending `splice` operation on it. To bound the number of nodes that are frozen but not finalized, we classify an unfinalized node as Type 0, 1, or 2, depending on the number of its subtrees that contain an unfinalized node. We show that each frozen, unfinalized node `X` of type 0 or 1 has a pending `remove` or `removeRec` at one of its descendants. So, there are $O(P \log L_{max})$ such nodes. We show that at most half of the unfinalized nodes are of type 2, so there are at most $L - R + O(P \log L_{max})$ type-2 nodes. Summing up yields the bound.

6 Memory Reclamation for Version Lists

We now describe how to safely reclaim the nodes spliced out of version lists and the Descriptor objects that are no longer needed. We apply an implementation of Reference Counting (RC) [2] with amortized expected $O(1)$ time overhead to a slightly modified version of our list. To apply RC in Figure 4, we add a reference count field to each Node or Descriptor and replace raw pointers to Nodes or Descriptors with reference-counted pointers. Reclaiming an object clears all its reference-counted pointers, which may lead to recursive reclamations if any reference count hits zero. This reclamation scheme is simple, but not sufficient by itself because a single pointer to a spliced out node may prevent a long chain of spliced out nodes from being reclaimed (see Figure 5, discussed later). To avoid this, we modify the `splice` routine so that whenever the `left` or `right` pointer of an node `Y` points to a descendant in the implicit tree, we set the pointer to $\top$ after `Y` is spliced out. Thus, only `left` and `right` pointers from spliced out nodes to their ancestors in the implicit tree remain valid. This ensures that there are only $O(\log L)$ spliced out nodes reachable from any spliced out node.

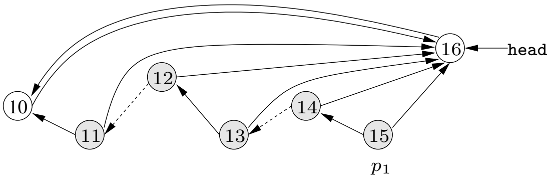


Figure 5 A portion of a version list where shaded nodes 15, 14, ..., 11 have been removed, in that order. Dotted pointers represent left and right pointers set to ⊤ by our modified `splice` routine. Node labels are counter values and vertical positioning represents nodes' priorities (cf. Figure 3).

This modification requires some changes to `find`. When a `find` reaches a node whose `left` pointer is ⊤, the traversal moves right instead; this results in following a valid pointer because whenever `splice(A, B, C)` is called, it is guaranteed that either `A` or `C` is an ancestor of `B`. For example in Figure 5, a process p_1, paused on node 15, will next traverse nodes 14, 16, and 10. Breaking up chains of removed nodes (e.g., from node 15 to 11 in Figure 5) by setting some pointers to ⊤ is important because otherwise, such chains can become arbitrarily long and a process paused at the head of a chain can prevent all of its nodes from being reclaimed. In the full version of the paper, we prove that traversing backwards does not have any significant impact on the time complexity of `find`. Intuitively, this is because backwards traversals only happen when the `find` is poised to read a node that has already been spliced out and each backwards traversal brings it closer to a non-removed node.

Using the memory reclamation scheme described above, we prove Theorems 11 and 12 that provide bounds similar to Theorems 9 and 10 in [8]. Both theorems include the resources needed by the RC algorithm, such as incrementing reference counts, maintaining retired lists, etc. Since the RC algorithm uses process-local hash tables, the amortized time bounds in Theorem 9 become amortized *in expectation* in Theorem 11. Using this scheme requires that `getHead` and `find` return reference counted pointers rather than raw pointers. Holding on to these reference counted pointers prevents the nodes that they point to from being reclaimed. For the space bounds in Theorem 12, we consider the number of reference counted pointers K, returned by version list operations that are still used by the application code. In most multiversioning systems (including the one in Appendix A), each process holds on to a constant number of such pointers, so $K \in O(P)$.

▶ **Theorem 11.** *The amortized expected time complexity of* `tryAppend`, `getHead`, `remove`, *and creating a new version list is* $O(1)$. *The amortized expected time complexity of* `find(V, ts)` *is* $O(n + \min(d, \log c))$, *where* n *is the number of version nodes with timestamp greater than* `ts` *that are reachable from* `V` *by following* `left` *pointers (measured at the start of the* `find`*),* d *is the depth of the VNode* `V` *in the implicit tree and* c *is the number of successful* `tryAppend` *from the time* `V` *was the list head until the end of the* `find`*. All operations are wait-free.*

▶ **Theorem 12.** *Assuming there are at most* K *reference-counted pointers to VNodes from the application code, at the end of any execution that contains* L *successful* `tryAppend` *operations,* R `remove` *operations and a maximum of* L_{max} *successful* `tryAppend`*s on a single version list, the number of VNodes and Descriptors that have been allocated but not reclaimed is* $O((L - R) + (P^2 + K) \log L_{max})$.

In RC, cycles must be broken before a node can be reclaimed. While there are cycles in our version lists, we show that VNodes that have been spliced out are not part of any cycle.

References

1 Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. StackTrack: An automated transactional approach to concurrent memory reclamation. In *Proc. 9th European Conference on Computer Systems*, pages 25:1–25:14, 2014. doi:10.1145/2592798.2592808.

2 Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. Concurrent deferred reference counting with constant-time overhead. In *Proc. 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 526–541, 2021.

3 James Aspnes, Hagit Attiya, and Keren Censor. Max registers, counters, and monotone circuits. In *Proc. 28th ACM Symposium on Principles of Distributed Computing*, pages 36–45, 2009.

4 Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

5 Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.

6 Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978. doi:10.1145/359460.359470.

7 Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. KiWi: A key-value map for scalable real-time analytics. *ACM Trans. Parallel Comput.*, 7(3):16:1–16:28, June 2020. doi:10.1145/3399718.

8 Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and time bounded multiversion garbage collection, 2021. Available from arXiv:2108.02775.

9 Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–86, 1976.

10 Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control–theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. doi:10.1145/319996.319998.

11 Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 104–117, 1999. doi:10.1145/301618.301648.

12 Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures*, pages 89–102, 2020. doi:10.1145/3350755.3400227.

13 Guy E. Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In *Proc. 34th International Symposium on Distributed Computing*, volume 179 of *LIPICS*, pages 5:1–5:17, 2020.

14 Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment*, 13(2):128–141, 2019.

15 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015. doi:10.1145/2767386.2767436.

16 Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: automatic lock-free memory reclamation. In *Proc. of the 26th ACM Symp. on Principles and Practice of Parallel Programming*, pages 205–218, 2021.

17 David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. In *Proc. 20th ACM Symposium on Principles of Distributed Computing*, pages 190–199, 2001. doi:10.1145/383962.384016.

18 Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proc. 30th ACM Symposium on Principles of Distributed Computing*, pages 99–108, 2011. doi:10.1145/1993806.1993821.

19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010. doi:10.1145/1835698.1835736.

20 Panagiota Fatourou and Nikolaos D Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

21 Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proc. 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019. doi:10.1145/3323165.3323197.

22 Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proc. 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–188, 2011. doi:10.1145/1941553.1941579.

23 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

24 Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 260–269, 2002.

25 Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. In *34th International Symposium on Distributed Computing*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

26 Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

27 Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer, 2002.

28 Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005. doi:10.1145/1062247.1062249.

29 Idit Keidar and Dmitri Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913 of *LNCS*, pages 150–165. Springer, 2015. doi:10.1007/978-3-319-14720-8_7.

30 Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. MV-RLU: Scaling read-log-update with multi-versioning. In *Proc. 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 779–792, 2019. doi:10.1145/3297858.3304040.

31 Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *Proc. Int. Conference on Distributed Computing and Networking*, pages 212–226, 2014.

32 Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in SAP HANA. In *Proc. International Conference on Management of Data*, page 1307–1318, 2016. doi:10.1145/2882903.2903734.

33 Li Lu and Michael L Scott. Generic multiversion STM. In *Proc. International Symposium on Distributed Computing*, pages 134–148. Springer, 2013.

34 M.M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004. doi:10.1109/TPDS.2004.8.

35 Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. Poster: Bundled references: An abstraction for highly-concurrent linearizable range queries. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 448–450, 2021.

36 Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

37 Ruslan Nikolaev and Binoy Ravindran. Snapshot-free, transparent, and robust memory reclamation for lock-free data structures. In *Proceedings of the 42nd ACM International Conference on Programming Language Design and Implementation*, pages 987–1002, 2021.

38 Christos H Papadimitriou and Paris C Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems*, 9(1):89–99, 1984.

39 Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective multi-versioning STM. In *Proc. International Symposium on Distributed Computing*, pages 125–140, 2011.

40 Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 16–25, 2010.

41 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras–non-blocking memory reclamation. In *Proc. 29th ACM Symp. on Parallelism in Algorithms and Architectures*, pages 367–369, 2017. doi:10.1145/3087556.3087588.

42 D. Reed. Naming and synchronization in a decentralized computer system. Technical Report LCS/TR-205, EECS Dept., MIT, 1978.

43 Niloufar Shafiei. Non-blocking doubly-linked lists with good amortized complexity. In *Proc. 19th Int. Conference on Principles of Distributed Systems*, volume 46 of *LIPIcs*, pages 35:1–35:17, 2015.

44 Julian Shun, Yan Gu, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *Proc. 26th ACM-SIAM Symposium on Discrete Algorithms*, pages 431–448, 2015.

45 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. NBR: Neutralization based reclamation. In *Proc. 26th ACM Symp. on Principles and Practice of Parallel Programming*, pages 175–190, 2021. doi:10.1145/3437801.3441625.

46 H. Sundell. Wait-free reference counting and memory management. In *Proc. 19th IEEE Symposium Parallel and Distributed Processing*, 2005. doi:10.1109/IPDPS.2005.451.

47 Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *J. Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

48 John D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

49 Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021. A full version is available from arXiv:2007.02372.

50 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proc. 23rd ACM Symp. on Principles and Practice of Parallel Programming*, pages 1–13, 2018. doi:10.1145/3178487.3178488.

51 Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. of the VLDB Endowment*, 10(7):781–792, 2017.

A Application to Snapshottable Data Structures

We present a summary of the multiversioning scheme of Wei et al. [49], and describe how the techniques in this paper can be applied to achieve good complexity bounds.

The Multiversioning Scheme. Wei et al. [49] apply multiversioning to a concurrent data structure (DS) implemented from CAS objects to make it *snapshottable*. It does so by replacing each CAS object by a VersionedCAS object which stores a version list of all earlier values of the object. VersionedCAS objects support `vRead` and `vCAS` operations, which behave like ordinary read and CAS. They also support a `readVersion` operation which can be used to read earlier values of the object. Wei et al. present an optimization for avoiding the level of indirection introduced by version lists. For simplicity, we apply our MVGC technique to the version without this optimization.

Wei et al. also introduce a *Camera* object which is associated with these *VersionedCAS* objects. The Camera object simply stores a timestamp. A `takeSnapshot` operation applied to the Camera object attempts to increment the timestamp and returns the old value of the timestamp as a snapshot handle. To support read-only query operations on the concurrent DS (such as range-query, successor, filter, etc.), it suffices to obtain a snapshot handle s, and then read the relevant objects in the DS using `readVersion`(s) to get their values at the linearization point of the `takeSnapshot` that returned s. This approach can be used to add arbitrary queries to many standard data structures.

For multiversion garbage collection, Wei et al. [49] uses a variation of EBR [23], inheriting its drawbacks. Applying our range-tracking and version-list data structures significantly reduces space usage, resulting in bounded space without sacrificing time complexity.

Applying Our MVGC Scheme. Operations on snapshottable data structures (obtained by applying the technique in [49]) are divided into *snapshot queries*, which use a snapshot handle to answer queries, and *frontier operations*, which are inherited from the original non-snapshottable DS. We use our doubly-linked list algorithm (with the memory reclamation scheme from Section 6) for each VersionedCAS object's version list, and a range-tracking object `rt` to announce timestamps and keep track of required versions by ongoing snapshot queries. We distinguish between objects inherited from the original DS (DNodes) and version list nodes (VNodes). For example, if the original DS is a search tree, the DNodes would be the nodes of the search tree. See [8] for the enhanced code of [49] with our MVGC scheme.

At the beginning of each snapshot query, the taken snapshot is announced using `rt.announce()`. At the end of the query, `rt.unannounce()` is called to indicate that the snapshot that it reserved is no longer needed. Whenever a `vCAS` operation adds a new VNode `C` to the head of a version list, we `deprecate` the previous head VNode `B` by calling `rt.deprecate(B, B.timestamp, C.timestamp)`. Our announcement scheme prevents VNodes that are part of any ongoing snapshot from being returned by `deprecate`.

Once a VNode is returned by a `deprecate`, it is removed from its version list and the reclamation of this VNode and the Descriptors that it points to is handled automatically by the reference-counting scheme of Section 6. Thus, we turn our attention to DNodes. A DNode can be reclaimed when neither frontier operations nor snapshot queries can access it.

We assume that the original, non-snapshottable DS comes with a memory reclamation scheme, MRS, which we use to determine if a DNode is needed by any frontier operation. We assume that this scheme calls `retire` on a node X when it becomes unreachable from the roots of the DS, and `free` on X when no frontier operations need it any longer. This assumption is naturally satisfied by many well-known reclamation schemes (e.g., [28, 41, 23]).

Even when MRS `free`s a DNode, it may not be safe to reclaim it, as it may still be needed by ongoing snapshot queries. To solve this problem, we tag each DNode with a birth timestamp and a retire timestamp. A DNode's birth timestamp is set after a DNode is allocated but before it is attached to the data structure. Similarly, a DNode's retire timestamp is set when MRS calls `retire` on it. We say that a DNode is *necessary* if it is not yet freed by MRS, or if there exists an announced timestamp in between its birth and retire timestamp. We track this using the same range-tracking data structure `rt` that was used for VNodes. Whenever MRS `frees` a DNode `N`, we instead call `rt.deprecate(N, N.birthTS, N.retireTS)`. When a DNode gets returned by a `deprecate`, it is no longer needed so we reclaim its storage space.

We say that a VNode is *necessary* if it is pointed to by a DNode that has not yet been deprecated (i.e. freed by MRS) or if its interval contains an announced timestamp. Let D and V be the maximum, over all configurations in the execution, of the number of necessary DNodes and VNodes, respectively. Theorem 13 bounds the overall memory usage of our memory-managed snapshottable data structure. Theorem 14 is an amortized version of the time bounds proven in [49].

▶ **Theorem 13.** *Assuming each VNode and DNode takes $O(1)$ space, the overall space usage of our memory-managed snapshottable data structure is $O(D + V + P^2 \log P + P^2 \log L_{max})$, where L_{max} is the maximum number of successful* `vCAS` *operations on a single* `VCAS` *object.*

▶ **Theorem 14.** *A snapshot query takes amortized expected time proportional to its sequential complexity plus the number of* `vCAS` *instructions concurrent with it. The amortized expected time complexity of frontier operations is the same as in the non-snapshottable DS.*

A Tight Local Algorithm for the Minimum Dominating Set Problem in Outerplanar Graphs

Marthe Bonamy
CNRS, LaBRI, Université de Bordeaux, France

Linda Cook
Discrete Mathematics Group, Institute for Basic Science (IBS), Daejeon, Republic of Korea

Carla Groenland
Utrecht University, The Netherlands

Alexandra Wesolek
Department of Mathematics, Simon Fraser University, Burnaby, Canada

Abstract

We show that there is a deterministic local algorithm (constant-time distributed graph algorithm) that finds a 5-approximation of a minimum dominating set on outerplanar graphs. We show there is no such algorithm that finds a $(5-\varepsilon)$-approximation, for any $\varepsilon > 0$. Our algorithm only requires knowledge of the degree of a vertex and of its neighbors, so that large messages and unique identifiers are not needed.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Approximation algorithms

Keywords and phrases Outerplanar graphs, dominating set, LOCAL model, constant-factor approximation algorithm

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.13

Related Version *Full Version*: https://arxiv.org/abs/2108.02697

Funding *Marthe Bonamy*: Supported by the ANR project GrR (ANR-18-CE40-0032).
Linda Cook: This work was supported by AFOSR grant A9550-19-1-0187, NSF grant DMS-1800053 and the Institute for Basic Science (IBS-R029-C1).
Carla Groenland: Supported by the European Research Council Horizon 2020 project CRACKNP (grant agreement no. 853234).
Alexandra Wesolek: Supported by the Vanier Canada Graduate Scholarships program.

Acknowledgements We thank the referees for helpful comments which improved the presentation of the paper.

1 Introduction

Given a sparse graph class, how well can we approximate the size of the minimum dominating set (MDS) in the graph using a constant number of rounds in the LOCAL model? A *dominating set* of a graph $G = (V, E)$ is a set $S \subseteq V$ such that every vertex in $V \setminus S$ has a neighbor in S. Given a graph G and an integer k, deciding whether G has a dominating set of size at most k is NP-complete even when restricting to planar graphs of maximum degree three [9]. Moreover, the size of the MDS is NP-hard to approximate within a constant factor (for general graphs) [16]. The practical applications of MDS are diverse but almost always involve large networks [3], and it is therefore natural to turn to the the distributed setting. No constant factor approximation of the MDS is possible using a sub-linear number of rounds in the LOCAL model [13], and so various structural restrictions have been considered on the graph classes with the hope of finding more positive results (see [8] for an overview).

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 13; pp. 13:1–13:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Planar graphs are a hallmark case. For planar graphs, guaranteeing that some constant factor approximation can be achieved is already highly non-trivial [7, 14]. The current best known upper-bound is 52 [19], while the best lower-bound is 7 [11]. Substantial work has focused on generalizing the fact that some constant factor approximation is possible to more general classes of sparse graphs, like graphs that can be embedded on a given surface, or more recently graphs of bounded expansion [1, 2, 5, 12]. Tight bounds currently seem out of reach in those more general contexts.

In this paper we focus instead on restricted subclasses of planar graphs. Better approximation ratios can be obtained with additional structural assumptions: 32 if the planar graph contains no triangle [3] and 18 if the planar graph contains no cycle of length four [4]. These bounds are not tight, and in fact we expect they can be improved significantly. We are able to provide tight bounds for a different type of restriction: we consider planar graphs with no $K_{2,3}$-minor or K_4-minor[1], i.e. *outerplanar* graphs. Outerplanar graphs can alternatively be defined as planar graphs that can be embedded so that there is a special face which contains all vertices in its boundary.

Outerplanar graphs are a natural intermediary graph class between planar graphs and forests. A planar graph on n vertices contains at most $3n - 6$ edges, and a forest on n vertices contains at most $n - 1$ edges; an outerplanar graph on n vertices contains at most $2n - 3$ edges. Every planar graph can be decomposed into three forests [15]; it can also be decomposed into two outerplanar graphs [10].

For planar graphs, as discussed above, we are far from a good understanding of how to optimally approximate Minimum Dominating Set in $O(1)$ rounds. Let us discuss the case of forests, as it is of very relevant to the outerplanar graph case. For forests, a trivial algorithm yields a 3-approximation: it suffices to take all vertices of degree at least 2 in the solution, as well as vertices with no neighbor of degree at least 2 (that is, isolated vertices and isolated edges). The output is clearly a dominating set, and the proof that it is at most three times as big as the optimal solution is rather straightforward. In fact, the trivial algorithm is tight because of the case of long paths. Indeed, no constant-time algorithm can avoid taking all but a sub-linear number of vertices of a long path, while there is a dominating set containing only a third of the vertices.

Our contribution

We prove that a similarly trivial algorithm (as the one described for forests above) works to obtain a 5-approximation of MDS for outerplanar graphs in the LOCAL model.

Algorithm 1 A local algorithm to compute a dominating set in outerplanar graphs.

```
Input: An outerplanar graph $G$
Result: A set $S \subseteq V(G)$ that dominates $G$
In the first round, every vertex computes its degree and sends it to its neighbors;
$S := \{\text{Vertices of degree} \geq 4\} \cup \{\text{Vertices with no neighbor of degree} \geq 4\}$;
```

It is easy to check that the algorithm indeed outputs a dominating set. It is significantly harder to argue that the resulting dominating set is at most 5 times as big as one of minimum size. To do that, we delve into a rather intricate analysis of the behavior of a hypothetical counterexample, borrowing tricks from structural graph theory (see Lemma 2).

The proof that the bound of 5 is tight for outerplanar graphs is similar to the proof that the bound of 3 is tight for trees. Every graph in the family depicted in Figure 1 is outerplanar, and every local algorithm that runs in a constant number of rounds selects all

[1] For any integer $n \geq 1$, K_n denotes the complete graph on n vertices. For integers $n, m \geq 1$, $K_{n,m}$ denotes the complete bipartite graph with partite classes of size n and m.

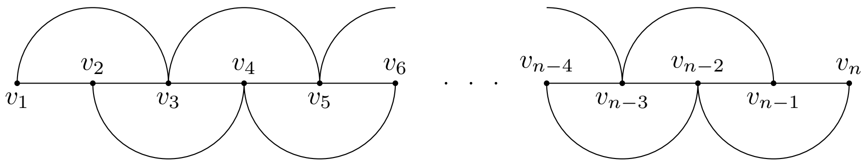


Figure 1 The graph G_n^- is a path $v_1, \dots, v_n$ to which we add all edges between vertices of distance two. In this example n is even.

but a sub-linear number of vertices [7, pp. 87–88]. Informally, all but a sub-linear number of vertices "look the same" – see Section 3 for more details, and [17] for an excellent survey of lower bounds.

Our main result is the following.

▶ **Theorem 1.** *There is an algorithm that computes a* 5*-approximation of Minimum Dominating Set for outerplanar graphs in* $O(1)$ *rounds in the LOCAL model. This is tight, in the sense that no algorithm can compute a* $(5 - \varepsilon)$*-approximation with the same constraints, for any* $\varepsilon > 0$.

In other words, there is a trivial local algorithm for Minimum Dominating Set in outerplanar graphs that turns out to be tight. All the difficulty lies in arguing that the approximation factor is indeed correct.

We note that the algorithm is so trivial that every vertex only needs to send one bit of information to each of its neighbors ("I have degree at least 4" or "I have degree at most 3"). The network might be anonymous – names are not useful beyond being able to count the number of neighbors, and the solution is extremely easy to update when there is a change in network. For contrast, in anonymous planar graphs the best known approximation ratio is 636 [18].

It is important to note that there is no hope for such a trivial algorithm in the case of planar graphs. Indeed, in Figure 2, we can see that for any p, no algorithm taking all vertices of degree $\geq p$ in the solution can yield a constant-factor approximation in planar graphs. However, the case of outerplanar graphs shows that the road to a better bound for planar graphs might go through finer structural analysis rather than smarter algorithms.

Definitions and notation

For a vertex set $A \subseteq V$, let $G[A]$ denote the induced subgraph of G with vertex set A. Let $E(A)$ denote set of edges of $G[A]$. For vertex sets $A, B \subseteq G$, let $E(A, B)$ denote the set of edges in G with one end in A and the other end in B. We write $G \setminus e$ for the graph in which the edge e is removed from the edge set of G. For a set $P \subseteq V$ inducing a connected subgraph, we write G/P for the graph obtained by contracting the set P: we replace the vertices in P with a new vertex v_P, which is adjacent to $u \in V \setminus P$ if and only if u has some neighbor in P. For a set X of vertices, we let $N[X]$ denote the set $X \cup \cup_{x \in X} N(x)$ and we let $N(X)$ denote the set $N[X] \setminus X$. If $x_1, x_2, \dots, x_k$ are the elements of X, we may also denote $N[X]$ and $N(X)$ as $N[x_1, x_2, \dots, x_k]$ and $N(x_1, x_2, \dots, x_k)$, respectively.

Given a graph G, let $V_{4+}(G)$ denote the set of vertices of degree at least 4 in G, and let $V^*(G)$ denote the set $V(G) \setminus N[V_{4+}(G)]$. In other words, $V^*(G)$ is the set of vertices of degree at most 3 in G which only have neighbors of degree at most 3. For a graph G and a dominating set S of G, we denote $V_{4^+}(G) \setminus S$ by $B_S(G)$ and we denote $V^*(G) \setminus S$ by $D_S(G)$. We additionally let $A_S(G)$ denote the set $V(G) \setminus (S \cup D_S(G) \cup B_S(G))$. In situations where

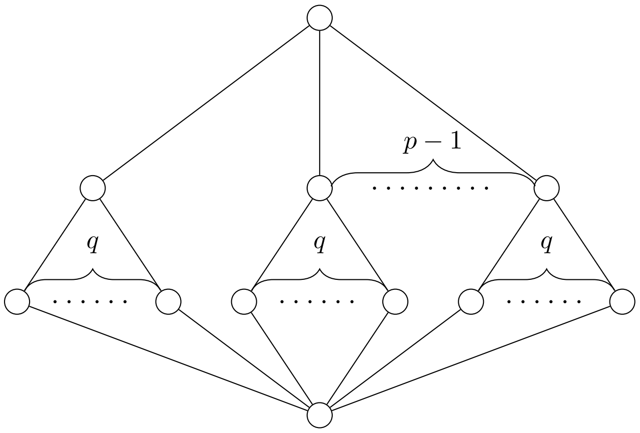


Figure 2 For any $p, q \in \mathbb{N}$, there is a planar graph $G_{p,q}$ which admits a dominating set of size 2 such that $|\{\text{Vertices with degree } \geq q\}| \geq p$.

our choice of G, S is not ambiguous we will simply write B, D, A for $B_S(G)$, $D_S(G)$ and $A_S(G)$, respectively. An overview of the notation is given in Table 1.

Table 1 An overview of the notation used in Section 2.

v is an element of	$\deg(v)$	degrees of neighbors of v	further restrictions
$V_{4^+}(G)$	≥ 4	arbitrary	-
$B_S(G)$	≥ 4	arbitrary	$v \notin S$
$V^*(G)$	≤ 3	≤ 3	-
$D_S(G)$	≤ 3	≤ 3	$v \notin S$
$A_S(G)$	≤ 3	at least one neighbor of degree ≥ 4	$v \notin S$

An outerplanar embedding of G is an embedding in which a special *outer face* contains all vertices in its boundary.

We denote by $H_G(S)$ the multigraph with vertex set S, obtained from G as follows. For every vertex u in $V(G) \setminus S$, we select a neighbor $s(u) \in N(u) \cap S$, and contract the edge $\{u, s(u)\}$. Contrary to the contraction operation mentioned earlier, this may create parallel edges, but we delete all self-loops. The resulting multigraph inherits the set S as its vertex set. We refer to Figure 3 for an example.

Note that $H_G(S)$ inherits an outerplanar embedding from G. If the graph G and the dominating set S are clear, we will write H for $H_G(S)$. Lemma 3 provides some intuition as to why the graph H is useful.

Properties of outerplanar graphs

Here we mention some standard but useful properties of outerplanar graphs. A graph H is a *minor* of a graph G if H can be obtained from G through a series of vertex or edge deletions and edge contractions. Alternatively, an H-minor of G consists of a connected set $X_h \subseteq V(G)$ for each $h \in V(H)$ and a set of paths $\{P_{hh'} | hh' \in E(H)\}$, where $P_{hh'}$ is a path in G between a vertex in X_h and a vertex in $X_{h'}$, all of which are pairwise vertex-disjoint except for possibly their ends. Note that any minor of an outerplanar graph is outerplanar. Neither K_4 nor $K_{2,3}$ can be drawn in the plane so that all vertices appear on the boundary of a special face. Therefore, outerplanar graphs are K_4-minor-free and $K_{2,3}$-minor-free.

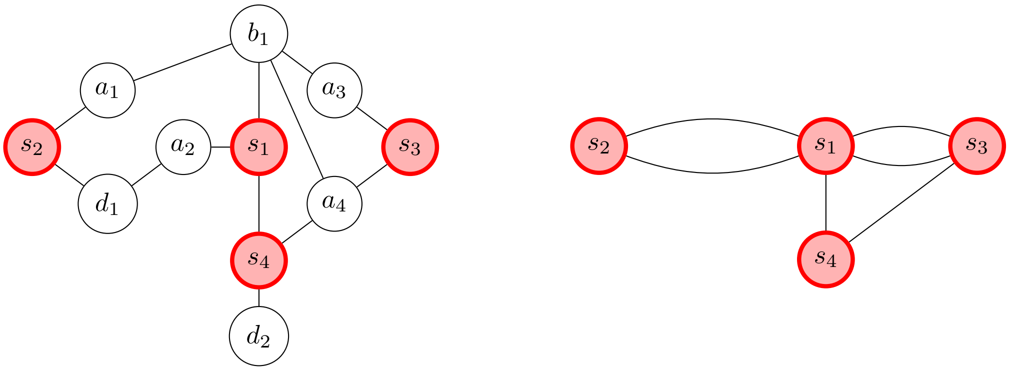


Figure 3 On the left is a graph G with dominating set $S = \{s_1, s_2, s_3, s_4\}$. The vertex $s(u)$ is uniquely determined for all $u \neq a_4$. On the right is the graph $H_G(S)$ for $s(a_4) = s_3$.

Any outerplanar graph G satisfies $|E(G)| \leq 2|V(G)| - 3$ by a simple application of Euler's formula. It follows immediately that every outerplanar graph contains a vertex of degree at most 3, but a standard structural analysis guarantees that every outerplanar graph contains a vertex of degree at most 2.

2 Analysis of the approximation factor

This section is devoted to proving the following result. (An overview of the relevant notation is given in Table 1.)

▶ **Lemma 2.** *For every outerplanar graph G, any dominating set S of G satisfies $|S| \geq \frac{1}{4}(|B_S(G)| + |D_S(G)|)$.*

We briefly argue that Lemma 2 yields the desired result. Given an outerplanar graph, Algorithm 1 outputs $V_{4^+}(G) \cup V^*(G)$ as a dominating set. To argue that it is a 5-approximation of the Minimum Dominating Set problem, it suffices to prove that any dominating set S of G satisfies $|S| \geq \frac{1}{5}(|V_{4^+}(G) \cup V^*(G)|)$. For technical reasons, it is easier to bound S as a function of the vertices in $V_{4^+}(G) \cup V^*(G)$ that are not in S, i.e. $|S| \geq \frac{1}{4}(|B_S(G)| + |D_S(G)|)$, which yields $|S| \geq \frac{1}{5}(|V_{4^+}(G) \cup V^*(G)|)$.

We prove the lemma by analyzing the structure of a "smallest" counterexample. A counterexample satisfies

$$|S| < \frac{1}{4}(|B_S(G)| + |D_S(G)|),$$

and we will choose one which minimizes $|S|$ and with respect to that maximizes $|B_S(G)| + |D_S(G)|$. For this, we need that $|B_S(G)| + |D_S(G)|$ is bounded in terms of $|S|$ by some constant, otherwise a counterexample maximizing $|B_S(G)| + |D_S(G)|$ might not exist since $|B_S(G)| + |D_S(G)|$ could be arbitrarily large. We therefore first prove the following much weaker result.

▶ **Lemma 3.** *For every outerplanar graph G, any dominating set S of G satisfies $|S| \geq \frac{1}{39}(|B_S(G)| + |D_S(G)|)$.*

We did not try to optimize the constant 39 and rather aim to get across some of the main ideas as clearly as possible. The proof shows the importance of the graph $H_G(S)$, which we will also use in the proof of Lemma 2.

Proof of Lemma 3. We may assume that the graph G is connected; otherwise, we can repeat the same argument for each connected component of G. We fix an outerplanar embedding of G. For each $u \in V(G) \setminus S$ we select an arbitrary neighbor $s(u) \in N(u) \cap S$ that we contract it with (keeping parallel edges but removing self-loops), resulting in the multigraph $H_G(S)$ on the vertex set S. The key step in our proof is showing that $H_G(S)$ has bounded edge multiplicity. Indeed, every edge $s_1 s_2$ in $H_G(S)$ is obtained from G by contracting at least one vertex or from the edge $s_1 s_2$ in G. For $i \in \{1,2\}$, let V_i be the set of vertices contracted to s_i that gave an edge between s_1 and s_2 in $H_G(S)$. Since there is no $K_{2,3}$-minor in G (as G is outerplanar), we find $|V_1| \leq 2$ and $|V_2| \leq 2$. Any edge between s_1 and s_2 in $H_G(S)$ can now be associated with an edge between $\{s_1\} \cup V_1$ and $\{s_2\} \cup V_2$ in G, and hence edges in $H_G(S)$ have multiplicity at most 9 (this is far from tight).

We derive that $|E(H_G(S))| \leq 9|E(H')|$, where H' is the simple graph underlying $H_G(S)$ (i.e. the simple graph obtained by letting $s_1, s_2 \in S$ be adjacent in H' if and only if there is an edge between them in $H_G(S)$). Note that H' is a minor of G. Since outerplanar graphs are closed under taking minors, the graph H' is an outerplanar graph. It follows that $|E(H')| \leq 2|S| - 3$. Combining both observations, we get $|E(H_G(S))| \leq 18|S|$.

By outerplanarity, we have $|E(H_G(S))| \geq \frac{1}{2}|B_S(G)|$. Indeed, each vertex $u \in B_S(G)$ has at most two common neighbors with $s(u)$ (otherwise there would be a $K_{2,3}$), hence u has at least one neighbor v such that $v \notin N[s(u)]$. The edge uv corresponds to an edge in $E(H_G(S))$, hence each $u \in B_S(G)$ contributes at least half an edge to $E(H_G(S))$ (as v could be also in $B_S(G)$). We derive $\frac{1}{2}|B_S(G)| \leq |E(H_G(S))| \leq 18|S|$. We observe that $|D_S(G)| \leq 3|S|$: indeed, each vertex from S is adjacent to at most 3 vertices from $D_S(G)$, since any vertex adjacent to a vertex in $D_S(G)$ has degree at most 3 by definition. We conclude that $|B_S(G)| + |D_S(G)| \leq (36+3)|S| = 39|S|$. ◀

In Lemma 3 we use that edges in H have low multiplicity, from which we then obtain a bound on the size of S. In order to improve the bound from Lemma 3, we dive into a deeper analysis of the graph H.

Proof of Lemma 2. We will consider a special counterexample (G, S) (satisfying $|S| < \frac{1}{4}(|B_S(G)| + |D_S(G)|)$) so that our counterexample has a structure we can deal with more easily than a general counterexample. In particular we will choose a counterexample (G, S) amongst those that minimize S and with respect to that maximize $B_S(G) \cup D_S(G)$ to maximize and minimize certain other graph parameters.

Namely, we assume that (G, S) in order: minimizes $|S|$; maximizes $|B_S(G) \cup D_S(G)|$; minimizes $|E(B_S(G))|$; minimizes $|V(G)|$; maximizes $|E(S, N(S))|$; minimizes $|E(G)|$. Note that this is well-defined since we established $|B_S(G) \cup D_S(G)| \leq 39|S|$, and clearly $|E(S, N(S))| \leq |E(G)| \leq 2|V(G)|$ by outerplanarity. Consequently, if a counterexample exists, then there exists one satisfying all of the above assumptions. More formally, we select a counterexample that is minimal for

$$(|S|, 39|S| - |B_S(G) \cup D_S(G)|, |E(B_S(G))|, |V(G)|, 2|V(G)| - |E(S, N(S))|, |E(G)|) \quad (\ddagger)$$

in the lexicographic order. Since all the elements in the sextuple are non-negative integers and their minimum is bounded below by zero, this is well-defined. (We remark that the parts indicated in gray were added to ensure the entries are non-negative; minimizing $39|S| - |B_S(G) \cup D_S(G)|$ comes down to maximizing $|B_S(G) \cup D_S(G)|$.)

While this approach is not entirely intuitive, the assumptions will prove to be extremely useful for simplifying the structure of G. For example, we can show that in a smallest counterexample that minimizes (‡), S is a stable set (Claim 5) and no two vertices in S have a common neighbor (Claim 7). In general, the Claims 4 to 14 show that such a minimal counterexample G has strong structural properties, by showing that otherwise we could delete some vertices and edges, or contract edges, and find a smaller counterexample.

Informally, for any vertex from S that we remove from the graph, we may decrease $|B_S(G) \cup D_S(G)|$ by 4 while maintaining $|S| < \frac{1}{4}(|B_S(G) \cup D_S(G)|)$. It is therefore natural to consider what happens when we reduce $|S|$ by one by contracting a connected subset containing two or more vertices from S. The result is again an outerplanar graph and we aim to show it is a smaller counterexample (unless the graph has some nice structure). Contracting an edge uv can affect the degrees of the remaining vertices in the graph G. Therefore $B_S(G)$ may "lose" additional vertices besides u and v if more than one of its neighbors are contracted and $D_S(G)$ may "lose" additional vertices if a neighbor got contracted, increasing the degree. We remark that vertices from $D_S(G)$ have no neighbors in $B_S(G)$, and therefore removing or contracting them does not affect the set $B_S(G)$.

We note that our minimal counterexample (G, S) is connected and again fix an outerplanar embedding of G. By definition, vertices in D can have no neighbors in B. In fact, the following stronger claim holds.

▷ Claim 4. Every vertex $d \in D$ satisfies $N(d) \subseteq S$.

Proof. Let $e = dv \in E(G)$ be such that $d \in D$. Suppose $v \notin S$. We consider the graph $G \setminus e$. Since $v \notin S$, S is a dominating set of $G \setminus e$. We find $|B_S(G \setminus e)| = |B_S(G)|$, since a vertex in D has no neighbor in B. Similarly, $|D_S(G \setminus e)| = |D_S(G)|$. Hence we also find that $|S| < \frac{1}{4}(|B_S(G \setminus e)| + |D_S(G \setminus e)|)$. Since $v \notin S$, the number of edges with one end incident to S is the same in G and $G \setminus e$. It follows that $(G \setminus e, S)$ is a counterexample to Lemma 2. Since $v \notin S$, G and $G \setminus e$ have the same number of edges with exactly one end in S. Hence since $|E(G \setminus e)| < |E(G)|$, the pair $(G \setminus e, S)$ is smaller with respect to ‡, contradicting our choice of (G, S). ◁

We are now ready to make more refined observations about the structure of (G, S). When considering a pair (G', S') that is smaller than (G, S) with respect to ‡ with $V(G') \subseteq V(G)$, it can be useful to refer informally to vertices that belong to $B_S(G)$ but not to $B_{S'}(G')$ as *lost* vertices (similarly for $D_S(G)$ and $D_{S'}(G')$). The number of lost vertices is an upper bound on $|B_S(G) \cup D_S(G)| - |B_{S'}(G') \cup D_{S'}(G')|$.

We need the following notation. Let $P \subseteq E(G)$. We denote the multigraph obtained from G by contracting every edge in P and deleting self-loops by G/P. Note G/P remains outerplanar and may contain parallel edges.

▷ Claim 5. The set S is a stable set.

Proof. Assume towards a contradiction that there are two vertices u and w in S that are adjacent.

Consider the outerplanar graph $G' = G/\{uw\}$ and let v_{uw} be the vertex resulting from the contraction of the edge uw. Let $S' = S \setminus \{u, w\} \cup \{v_{uw}\}$. Define $B' = B_{S'}(G')$, and $D' = D_{S'}(G')$. Note that S' dominates G'. Since we reduced the size of the dominating set by one, we are allowed to "lose up to four vertices from $B \cup D$", as then we get that

$$|B'| + |D'| \geq |B| + |D| - 4 > 4(|S| - 1) = 4|S'|.$$

We will now show the above inequality holds. If $v \in B \setminus B'$, then v is a common neighbor of u and w (and u, w have at most two such neighbors by outerplanarity). If $v \in D \setminus D'$, then v is a neighbor of u and/or w in G (since no vertex in $V(G) \cap V(G')$ has a higher degree in G' than G). We consider three cases, depending on the neighbors of u and of w in D.

- Suppose that u, w have no neighbors in D. Then $D' = D$ and we lose only vertices from B which are common neighbors of u, w, so at most 2.
- Suppose that u, w both have neighbors in D. Then u, w are of degree at most 3. Since they are adjacent to each other, they are adjacent to at most 4 other vertices in total. So $|B'| + |D'| \geq |B| + |D| - 4$.
- Suppose that u has a neighbor in D and w does not (the other case is analogous). Now $|B| + |D|$ decreases by at most 2, since any "lost" vertex is adjacent to u and distinct from w.

In all cases, S has decreased by 1 and $|B| + |D|$ by at most 4, so we indeed find $|B'| + |D'| > 4|S'|$. Since $|S'| < |S|$, this gives a contradiction with the minimality of our choice of (G, S). ◁

We remark that no vertex in S has only neighbors in D. Indeed, if some $s \in S$ has only neighbors in D then we remove s and its neighborhood from the graph and we have a smaller counterexample, as we reduced $|S|$ by one and $|D|$ by at most 3. In fact, we will show the following:

▷ Claim 6. If $d \in D$ and $s \in N(d) \cap S$, then s has a neighbor in B.

Proof. Suppose that $d \in D$ is adjacent to $s \in S$. Since $d \in D$, we find that s has degree at most 3. Say s has neighbors w_1 and w_2 (possibly equal, but both not equal to d). We argued above that s has a neighbor outside of D, so without loss of generality $w_1 \notin D$. Suppose towards a contradiction that $w_1, w_2 \notin B$. By Claim 5, we find $w_1, w_2 \notin S$, and so w_1, w_2 have degree at most 3. As $w_1 \notin D$, we find $w_1 \in A$.

Suppose first that w_1 and w_2 together have at most two neighbors outside of $\{w_1, w_2, s\}$. When we remove $N[s]$ from the graph, $|S|$ goes down by one and $|B \cup D|$ goes down by at most four ('counting' the two outside neighbors, w_2 and d), contradicting the minimality of our counterexample. So w_1, w_2 have at least three "outside" neighbors, which implies that w_2 exists and that w_1, w_2 are non-adjacent (see Figure 4). Moreover, w_1, w_2 together have at least three neighbors in B by using the same strategy (showing that deleting $N[s]$ would give a smaller counterexample). Since w_1 has degree at most 3 and is already adjacent to s, it has at most two neighbors in B. Thus w_2 is also not in D, since vertices in D have no neighbors in B.

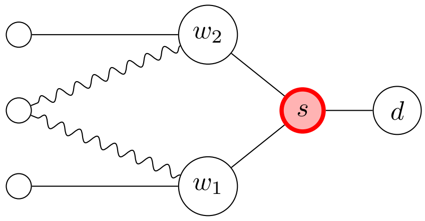


Figure 4 An illustration of the case when w_1, w_2 are not in B and together have three neighbors which are not s, w_1 or w_2. At least one of the wavy edges is present and at least one of w_1, w_2 has degree three in the picture. In particular, w_1 and w_2 are not adjacent.

Recalculating now that we know that $w_1, w_2 \notin B \cup D$, if $N(\{w_1, w_2\})$ contains at most three vertices of degree four in B, then $|B \cup D|$ goes down by at most four in $G \setminus N[s]$. This would be a contradiction as $(G, \setminus N[s], S \setminus \{s\})$ would be a smaller counterexample. Hence, w_1, w_2 have exactly four neighbors in B, all of which are of degree exactly 4.

We show that we may assume that d has degree 1. If d has another neighbor, it is in S by Claim 4. We delete the vertices s, w_1, w_2; since d is still in D and dominated, $|B \cup D|$ decreases by at most 4, and hence $(G \setminus \{s, w_1, w_2\}, S \setminus s)$ is a smaller counterexample. Hence we can assume d has degree one and therefore the edge ds is on the outer face.

Let G' be the graph obtained from G by adding the edge w_1w_2. Considering the local rotation of the three neighbors of s in an outerplanar embedding of G, we note that w_1, w_2 are consecutive neighbors of s. We can draw the edge w_1w_2 close to the path w_1-s-w_2 keeping the embedding outerplanar (note that the edge ds is still on the outer face). It follows that $w_1, w_2 \in B_S(G') \setminus B_S(G)$, so $|B_S(G) \cup D_S(G)| < |B_S(G') \cup D_S(G')|$. Thus, (G', S) is a smaller counterexample, a contradiction. ◁

▷ Claim 7. No two vertices in S have a common neighbor.

Proof. Assume towards a contradiction that there are two vertices s_1 and s_2 in S that have a common neighbor v. Since S is a stable set (Claim 5), we have $v \notin S$.

We will consider the outerplanar graph $G' = G/P$ obtained by contracting $P = \{s_1v, vs_2\}$ into a single vertex v_P. Let $S' = S \setminus \{s_1, s_2\} \cup \{v_P\}$. We use the abbreviations $B' = B_{S'}(G')$ and $D' = D_{S'}(G')$.

We will again do a case analysis, on the union of the neighbors of s_1 and the neighbors of s_2 in $D \setminus \{v\}$, to find a smaller counterexample. If $|B'| + |D'| \geq |B| + |D| - 4$, then (G', S') is a smaller counterexample. Note that vertices in $B \setminus B'$ have at least two neighbors in the set $\{s_1, v, s_2\}$.

- Suppose first that for some $i \in \{1, 2\}$, s_i is adjacent to at least two vertices in $D \setminus \{v\}$. Then v is the only other neighbor of s_i, so the graph G'' obtained from G by deleting s_i and its two neighbors in D, satisfies $|B(G'') \cup D(G'')| \geq |B \cup D| - 3$ whereas the set $S'' = S \setminus \{s_i\}$ is dominating. This gives a smaller counterexample.
- Suppose that both s_1, s_2 are adjacent to a single vertex in $D \setminus \{v\}$. Then both have degree at most 3. Let $d_1, d_2 \in D \setminus \{v\}$ be the neighbors of s_1, s_2 respectively (where d_1, d_2 might be equal). The graph G' is a smaller counterexample unless we lost two vertices from B besides possibly v, that is, $|B'| \leq |B \setminus \{v\}| - 2$. Any vertex lost from $B \setminus \{v\}$ must be adjacent to two vertices among $\{s_1, v, s_2\}$ (as otherwise its degree did not change), and since both s_1 and s_2 already have two named neighbors, G' is a counterexample unless there is, for each $i \in \{1, 2\}$, a common neighbor $b_i \in B$ of s_i and v, and all named vertices are distinct.

  Since $d_1 \in D$ and $b_1 \in B$, we find that b_1d_1 is not an edge of G. Since s_1 has three neighbors, b_1 and d_1 are consecutive neighbors and the edge b_1d_1 can be added without making the graph non-planar. Consider adding the edge b_1d_1 in G along the path $b_1s_1d_1$, such that there are no vertices in between the edge and the path. This may affect whether s_1 is on the outer face, but it does not affect whether s_2 is on the outer face. Therefore, after contracting this adjusted graph, the obtained graph G'' is still outerplanar. Moreover, b_1 has the same degree in G'' as in G, and so $|B_{S'}(G'') \cup D_{S'}(G'')| \geq |B| + |D| - 4$ and G'' is a smaller counterexample.
- Suppose that s_1 is adjacent to a vertex d_1 in $D \setminus \{v\}$ and s_2 is not (the symmetric case is analogous). There can be at most three vertices in $B \setminus \{v\}$ which are adjacent to two vertices in s_1, v, s_2 (as only one can be adjacent to s_1 and v, s_2 have at most two common neighbors since the graph is outerplanar). The only way in which G' is not a counterexample, is when there is a common neighbor b_1 of s_1 and v and two common neighbors b_2, b_3 of s_2 and v with all named vertices distinct. As before, we may now add the edge b_1d_1 in order to obtain a smaller counterexample G''.

- Finally, suppose that s_1 and s_2 have no neighbors in $D \setminus \{v\}$. By outerplanarity, there are at most four vertices with two neighbors among $\{s_1, v, s_2\}$. Hence G' is a counterexample unless there are exactly four (the only vertices "lost" from $B \cup D$ are either v or among such common neighbors, since s_1 and s_2 have no neighbors in $D \setminus \{v\}$). All four vertices are adjacent to v, because otherwise G contains a $K_{2,3}$-minor[2], a contradiction. In particular, G' is a counterexample unless there are two common neighbors of v and s_1 and two common neighbors of v and s_2 (and so $d(v) \geq 6$ and $v \in B$).
  Fix a clockwise order $w_1, w_2, \dots, w_d$ on the neighbors of v such that the path $w_1 v w_d$ belongs to the boundary of the outer face. Let $i \neq j$ such that $w_i = s_1$ and $w_j = s_2$. After relabelling, we may assume $i < j$. Since s_1 and s_2 both have two common neighbors with v, we find $i > 1$, $j < d$ and $i + 1 < j - 1$. The vertices adjacent to multiple vertices in $\{s_1, v, s_2\}$ are $w_{i-1}, w_{i+1}, w_{j-1}$ and w_{j+1}. We create a new graph G'' by replacing v with two adjacent vertices v_1 and v_2, where v_1 is adjacent to $w_1, w_2, \dots, w_{i+1}$ and v_2 to $w_{i+2}, \dots, w_d$. This graph is outerplanar because both v_1 and v_2 have an edge incident to the outer face. Moreover, $d(v_1)$ and $d(v_2)$ are both at least 4, since they are adjacent to each other, to either s_1 or s_2 and to at least two vertices among $w_1, \dots, w_d$. The set S is still a dominating set, but $|B(G'') \cup D(G'')| > |B \cup D|$ so this is a smaller counterexample.

In all cases, we found a smaller counterexample. This contradiction proves the claim. ◁

Since vertices in D only have neighbors in S, the claim implies in particular that each vertex of D has degree 1.

With the claims above in hand, we now analyze the structure of $H = H_G(S)$ as described in the notation section more closely. Note that the for each $u \in V(G) \setminus S$ the vertex $s(u)$ is uniquely defined by Claim 7.

Recall that H is outerplanar. It follows that there is a vertex $s_1 \in V(H)$ with at most 2 distinct neighbors in H.

We start with an easy observation.

▶ **Observation 8.** *Let $b \in B$ and $s(b)$ be its unique neighbor in S. Then there exists $w \in N(b) \setminus \{s(b)\}$, such that its unique neighbor $s(w) \in S$ is not equal to $s(b)$.*

Indeed, the vertex b can have at most two common neighbors with $s(b)$ (otherwise there would be a $K_{2,3}$, contradicting outerplanarity), and a vertex in B has degree at least 4 by definition.

Note that the vertex s_1 has at least one neighbor in H. Indeed, if s_1 has no neighbor in H, then $N[s_1]$ is a connected component in G. Since G is connected, $G = N[s_1]$. By Observation 8, we have $B = \emptyset$, so $|D \cup B| \leq 3$.

▷ Claim 9. The vertex s_1 has precisely two neighbors in H.

Proof. Assume towards a contradiction that s_1 has a single neighbor s_2 in H. Let $v_1, \dots, v_k$ be the vertices in $N[s_1]$ that have a neighbor in $N[s_2]$, and conversely let $u_1, \dots, u_\ell$ be the vertices in $N[s_2]$ that have a neighbor in $N[s_1]$. Note that by Claims 5 and 7, all of $\{v_1, \dots, v_k, u_1, \dots, u_\ell, s_1, s_2\}$ are pairwise distinct. If $\ell \geq 3$, then contracting the connected set $N[s_1]$ in G gives a $K_{2,3}$ on the contracted vertex and s_2 on one side and u_1, u_2, u_3 on the other. We derive that $\ell \leq 2$, and by symmetry, $k \leq 2$. By Observation 8, the only neighbors of s_1 that belong to B are in $\{v_1, v_2\}$. As we assumed that s_1 has degree 1 in H, we have $N[v_i] \subseteq N[s_1] \cup \{u_1, u_2\}$ for $i \in \{1, 2\}$. We will do a case distinction on $N[s_1] \cap D$.

[2] The vertices s_1, s_2 can have at most one further common neighbor v^* besides v. If v^* exists, we contract it with s_1 and s_2. We find a $K_{2,3}$ subgraph with v, v^* on one side and the three other common neighbors on the other side.

- If s_1 has no neighbor in D, we delete $N[s_1]$, and note that $D_S(G) = D_{S\setminus s_1}(G \setminus N[s_1])$, while $B_S(G) \setminus \{v_1, v_2, u_1, u_2\} \subseteq B_{S\setminus s_1}(G \setminus N[s_1])$. Therefore,

$$|S \setminus \{s_1\}| \geq \frac{1}{4} \cdot (|D_{S\setminus s_1}(G \setminus N[s_1])| + |B_{S\setminus s_1}(G \setminus N[s_1])|).$$

  So we have found a smaller counterexample.
- Suppose s_1 has two neighbors $d_1 \neq d_2$ in D. Then v_2 does not exist since $d(s_1) \leq 3$ and because v_1, v_2 are distinct from d_1, d_2 (vertices in D have degree 1). Suppose first that v_1 has degree at least 4. Let x be its neighbor distinct from u_1, u_2, s_1. By assumption on s_1, the vertex x has no neighbor in $S \setminus \{s_1\}$. Therefore, x is adjacent to s_1. However, x is distinct from d_1, d_2 and v_1, which contradicts $d(s_1) \leq 3$. This case is illustrated in Figure 5. Hence $v_1 \notin B$ and removing $N[s_1]$ now gives a smaller counterexample, a contradiction.

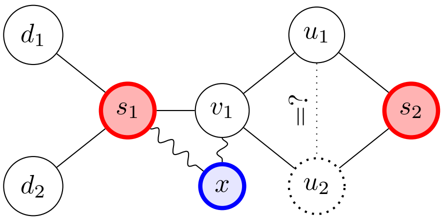


Figure 5 An illustration of the case where s_1 has degree one in H and two neighbors $d_1, d_2 \in D$ in G. If $v_1 \in B$, then some vertex x exists such that both wavy edges are present in G, a contradiction.

- Suppose that s_1 has a single neighbor d_1 in D. Removing $N[s_1]$ gives a smaller counterexample again, unless all of u_1, u_2, v_1, v_2 exist and belong to B. In particular, v_1, v_2 both have degree at least 4. Each of v_1 and v_2 can only have neighbors within $\{s_1, v_1, v_2, u_1, u_2\}$ because a neighbor x not within $\{s_1, v_1, v_2, u_1, u_2\}$ is a neighbor of s_1, but $d(s_1) \leq 3$. Therefore, both v_1 and v_2 are adjacent to u_1 and u_2. Together with s_1, this forms a $K_{2,3}$ subgraph (see Figure 6): a contradiction. ◁

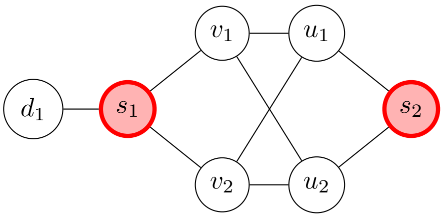


Figure 6 An illustration of the case where s_1 has degree one in H and has exactly one neighbor in D in G. We reduce to the case in which the depicted graph is a subgraph of G. We find a contradiction since the depicted graph contains a $K_{2,3}$.

.

So s_1 has two neighbors in H. Let $s_2, s_3 \in V(H)$ be its neighbors. In G, let $w_1, \ldots, w_p$ be the vertices in $N[s_1]$ that have a neighbor in $N[s_3]$, and conversely let $x_1, \ldots, x_q$ be the vertices in $N[s_3]$ that have a neighbor in $N[s_1]$. By the same argument as before for s_1 and s_2, we obtain $p \leq 2$ and $q \leq 2$ and that all of $\{w_1, w_2, x_1, x_2, s_1, s_3\}$ are pairwise distinct. However, there may be a vertex in $\{w_1, w_2\} \cap \{v_1, v_2\}$; there may not be two such vertices since this would lead to a $K_{2,3}$-minor (with vertices $\{v_1, w_1\}$ and $\{v_2, w_2\}$ in one part, and $s_1, \{s_2, u_1, u_2\}, \{s_3, x_1, x_2\}$ in the other).

Our general approach is to delete $N[s_1]$ and add edges between $\{u_1, u_2\}$ and $\{x_1, x_2\}$ as appropriate so as to mitigate the impact on $|B \cup D|$. If this does not work, we obtain further structure on the graph which we exploit to create a different smaller counterexample. We will repeatedly apply the following Observation 10. Sometimes when deleting vertices and edges from the graph G, the result is a disconnected graph, so we can perform the "flipping" operation described below, and connect the different components to get a smaller counterexample (G', S').

▶ **Observation 10** (Flipping). *Let G be the disjoint union of two outerplanar graphs O_1 and O_2. Consider an outerplanar embedding of G, and let $(u_1, u_2, \dots, u_q)$ denote the outer face of $G[O_2]$ in clockwise order. We can obtain a different outerplanar embedding of G by reversing the order of O_2 without modifying the embedding of O_1, so that the outer face of $G[O_2]$ is $(u_q, \dots, u_2, u_1)$ in clockwise order.*

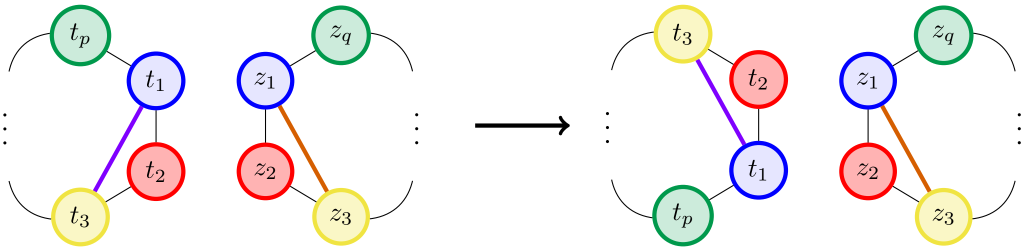


Figure 7 An illustration of Observation 10.

An example of the observation above is given in Figure 7. Beside Observations 8 and 10, the third useful observation is as follows.

▶ **Observation 11.** *$N[v_1, v_2, w_1, w_2] \subseteq N[s_1] \cup \{u_1, u_2, x_1, x_2\}$. Additionally, if $\{v_1, v_2\} \cap \{w_1, w_2\} = \emptyset$, then $N[v_1, v_2] \subseteq N[s_1] \cup \{u_1, u_2\}$ and $N[w_1, w_2] \subseteq N[s_1] \cup \{x_1, x_2\}$.*

This observation is argued similarly to Observation 8, we omit the argument.

Since s_1 is adjacent to s_2 and s_3 in H, all of u_1, x_1, v_1 and w_1 exist. We assume that either $\{v_1, v_2\} \cap \{w_1, w_2\} = \emptyset$ or $v_1 = w_1$. Note that $\{u_1, u_2\} \cap \{x_1, x_2\} = \emptyset$ since s_2 and s_3 do not have common neighbors by Claim 7. See Figure 8 for an illustration. For simplicity, when depicting which edges to add in which cases, we represent "u_2 does not exist" as "u_2 is possibly equal to u_1" (and variations). This means merely that if u_2 does not exist then the edges involving u_2 involve u_1 instead – multiple edges are ignored.

▷ Claim 12. One of w_2 and v_2 exists.

Proof. Suppose that neither w_2 nor v_2 exists. It is possible that $v_1 = w_1$, and that u_2 or x_2 do not exist. By Observation 11, if $v_1 \neq w_1$, then u_1, u_2 are not adjacent to w_1 and x_1, x_2 are not adjacent to v_1.

The degrees of x_1, x_2, u_1, u_2 in $G \setminus N[s_1]$ are at least one less than their degrees in G. Every vertex in $V(G) \setminus (N[s_1] \cup \{x_1, x_2, u_1, u_2\})$ has the same degree in G and in $G \setminus N[s_1]$. Let $S' = S \setminus \{s_1\}$, and note that S' dominates $G \setminus N[s_1]$.

- Suppose x_2, u_2 do not exist. If v_1 belongs to B, then it needs to have a neighbor which is not u_1, s_1 or one of x_1, w_1 (depending on whether $v_1 = w_1$), so it shares a neighbor with s_1 which is not in $B \cup D$. This implies that if $v_1 \in B$, then s_1 can have a neighbor

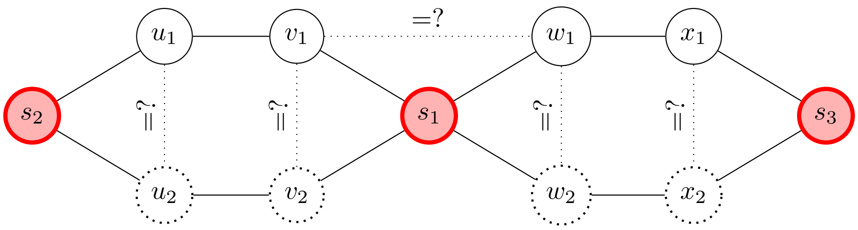


Figure 8 When s_1 has exactly two neighbors s_2, s_3 in H, each of s_2, s_3 has at most two neighbors with edges to vertices in $N[s_1]$. Moreover, s_2 and s_3 may have at most one common neighbor in $N[s_1]$. We draw vertices which may not exist in G as a dotted circle and connect vertices which may be equal with dotted edges. There may be more edges present in that are not drawn.

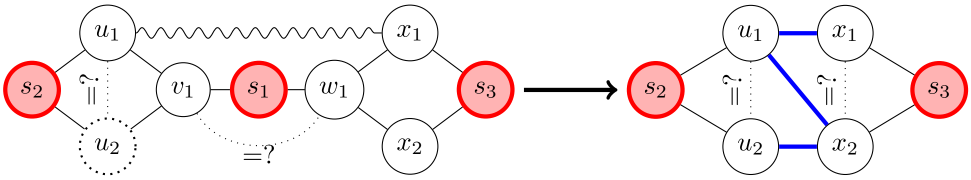


Figure 9 The case where w_2, v_2 do not exist. The original graph is drawn at the left and the modified graph is drawn at the right. The wavy line indicates there may be an edge between u_1 and x_1. Edges that may have been added are drawn in blue. Note that u_2 may not exist. There may be more edges which are not drawn (for instance v_1 might be adjacent to w_1) but these edges are not relevant to our argument.

$d_1 \in D$ or $w_1 \neq v_1$, but not both at the same time. It follows that regardless of whether $v_1 \in B$, we have $|N[s_1] \cap (B \cup D)| \leq 2$. But now $|(N[s_1] \cup \{x_1, u_1\}) \cap (B \cup D)| \leq 4$, so $(G \setminus N[s_1], S')$ is a smaller counterexample, a contradiction.

- By symmetry, we assume that x_2 exists. If u_1 and u_2 both exist, then they are not distinguishable at this point, which means we can swap their label. The same holds for x_1 and x_2. Hence we may assume that the vertices appear in the outer face in the order x_1, x_2, u_2, u_1, and that either u_1x_1 is an edge of G or there is no edge between $\{u_1, u_2\}$ and $\{x_1, x_2\}$. Let G' be the graph obtained from $G \setminus N[s_1]$ by adding the edges u_1x_1 (if it is not already present), u_1x_2 and (if u_2 exists) the edge u_2x_2 (see Figure 9). Note that G' is outerplanar and that S' dominates G'. Since G is outerplanar, if u_1x_1 is an edge in G, then neither u_1x_2 nor u_2x_2 is an edge in G. In G', the degrees of the vertices u_1, u_2, x_2 are at least as large as their respective degrees in G (the degree of x_1 might have dropped if the edge u_1x_1 was already present in G). Note that $|\{v_1, w_1, x_1\} \cup (N[s_1] \cap D)| \leq 4$, hence $|B_{S'}(G') \cup D_{S'}(G')| \geq |B \cup D| - 4$, a contradiction. ◁

▷ Claim 13. If $w_1 = v_1$, then v_2 and w_2 exist.

Proof. By Claim 12 we can assume v_2 exists. Suppose w_2 does not exist and $w_1 = v_1$. We remove $N[s_1]$ and add edges between $\{u_1, u_2\}$ and $\{x_1, x_2\}$ as above to ensure that for all but at most one of them, the degree does not decrease. To see an illustration of how the edges are added, see Figure 10. We suppose first that there are no edges between $\{u_1, u_2\}$ and $\{x_1, x_2\}$. The edges remedy the degree for x_1, x_2, since they only lost w_1, and for one of u_1, u_2; indeed, it is not possible that both u_1 and u_2 are adjacent to both v_1 and v_2 (since we would obtain a $K_{2,3}$ when considering s_1 as well).

By Observation 11, the degrees of other vertices are not affected by removing $N[s_1]$.

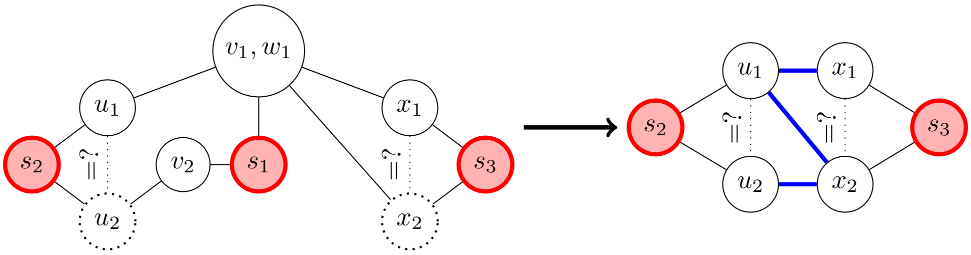


Figure 10 An example of the reduction for the case where v_1 is equal to w_1, v_2 exists and w_2 does not exist. We only draw edges which are relevant to our argument.

Again, since $|N[s_1] \cap (B \cup D)| \leq 3$, we have removed a vertex from S and at most 4 from $B \cup D$ so we have constructed a smaller counterexample.

We now assume $u_1 x_1$ is an edge.

- Assume that u_2 does not exist. Now v_2 has degree at most 3 unless it has a common neighbor with s_1, but then s_1 has no neighbor in D and we lose only two vertices from $N[s_1]$ and possibly u_1, x_1.
- Assume now that u_2 exists. Both x_1 and x_2 lose at most one edge, and we can ensure both gain at least one edge. So if we lose only two vertices from $N[s_1]$, plus possibly u_1, u_2, then we lose at most four vertices from $B \cup D$ in total. If there are three vertices from $B \cup D$ in $N[s_1]$, then v_2 is adjacent to both u_1, u_2 and s_1 has a neighbor d in D. We know that v_1 is adjacent to u_1 or u_2, but since there is the path $v_1 x_1 u_1$, we know that if v_1 would be adjacent to u_2, then there would be a $K_{2,3}$-minor with v_1, v_2 in one part and $\{u_1, x_1\}, u_2, s_1$ in the other part. Since u_2 is not adjacent to v_1, we know that u_2 loses at most one edge when deleting $N[s_1]$ and gains an edge when we add the edge $u_2 x_2$. This means we lose at most four vertices from $B \cup D$ (counting u_1, v_1, v_2 and d). ◁

We henceforth assume that $|\{v_1, v_2, w_1, w_2\}| \geq 3$. We can show that also in this case we can always reduce to a smaller counterexample.Since the details of the remaining casework are not particularly illuminating, we will omit them for brevity. Appendix A and the longer arXiv version [6] of our paper both contain the full details.

Since in all claims and cases we can show that there is a smaller counterexample, there can not be a counterexample to Lemma 2, which proves that Algorithm 1 computes a 5-approximation of Minimum Dominating Set for outerplanar graphs. ◀

3 Lower bound for outerplanar graphs

In this section we show that there is no deterministic local algorithm that finds a $(5 - \epsilon)$-approximation of a minimum dominating set on outerplanar graphs using T rounds, for any $\epsilon > 0$ and $T \in \mathbb{N}$. To do so we use a result from Czygrinow, Hańćkowiak and Wawrzyniak [7, pp. 87–88] who gave a lower bound in the planar case. For $n \equiv 0 \mod 10$, they consider a graph G_n, which is a cycle $v_1, v_2, \ldots, v_n, v_1$ where edges between vertices of distance two are added. They showed that for every local distributed algorithm $\mathcal{A}$ and every $\delta > 0$ and $n_0 \in \mathbb{N}$ there exists $n \geq n_0$ for which the algorithm $\mathcal{A}$ outputs a dominating set for G_n that is not within a factor of $5 - \delta$ of the optimal dominating set for G_n. Their graph G_n is not outerplanar, but we can delete three of its edges to get an outerplanar graph G_n^-. The graph G_n^- is a path $v_1 \ldots v_n$ where all edges between vertices of distance two are added as in Figure 1. The argument of [7] builds on a lower bound for local algorithms computing a maximum independent set, which in turn depends on multiple applications of Ramsey's theorem. A similar approach is used by [11] to obtain the best-known lower-bound for planar graphs. Using the graph G_n^-, this approach can also be used to prove our result; the main idea is that since in the middle all the vertices "look the same", no local algorithm can do better than selecting almost all of them.

Alternatively, we can exploit the result of [7] as follows. For any bound $T \in \mathbb{N}$ on the number of rounds, any vertex in $M = \{v_{2T+1}, \ldots, v_{n-2T-1}\}$ has the same local neighborhood in G_n as in G_n^-. Since G_n is rotation symmetric, a potential local algorithm also finds a dominating set D for G_n (for $n \geq 4T + 2$), and with the result of [7] we obtain $|D| \geq (5 - \delta)\gamma(G_n)$. For n sufficiently large with respect to T, the set D is the same as the set D' that the algorithm would give for G_n up to at most $\delta n/10$. Since $n \equiv 0 \mod 10$, $\gamma(G_n) = \gamma(G_n^-) = \frac{n}{5}$ and we find the desired lower-bound $|D'| \geq \left(5 - \frac{\delta}{2}\right)\gamma(G_n^-) \geq (5-\epsilon)\gamma(G_n^-)$ for δ small enough.

4 Conclusion

Through a rather intricate analysis of the structure of outerplanar graphs, we were able to determine that a very naive algorithm gives a tight approximation for minimum dominating set in outerplanar graphs in $O(1)$ rounds. While there are some highly non-trivial obstacles to extending such work to planar graphs, we believe that similar techniques can be used to vastly improve the state of the art for triangle-free planar graphs and for C_4-free planar graphs. In the first case, recall that a 32-approximation is known [3], and there is a simple construction (a large 4-regular grid) showing that 5 is a lower bound. We believe that 5 is the right answer. In the second case, an 18-approximation is known [4], and there is no non-trivial lower bound. We refrain from conjecturing the right bound here – we simply point out that there is no reason yet to think 3 is out of reach. We believe that very similar techniques to the ones developed here can be used to obtain a 9-approximation, and possibly lower.

References

1 Saeed Akhoondian Amiri, Patrice Ossona de Mendez, Roman Rabinovich, and Sebastian Siebertz. Distributed domination on graph classes of bounded expansion. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 143–151, 2018.

2 Saeed Akhoondian Amiri, Stefan Schmid, and Sebastian Siebertz. A local constant factor mds approximation for bounded genus graphs. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, page 227–233, 2016.

3 Sharareh Alipour, Ehsan Futuhi, and Shayan Karimi. On distributed algorithms for minimum dominating set problem, from theory to application, 2021. `arXiv:2012.04883`.

4 Sharareh Alipour and Amir Jafari. A local constant approximation factor algorithm for minimum dominating set of certain planar graphs. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, pages 501–502, 2020.

5 Saeed Akhoondian Amiri, Stefan Schmid, and Sebastian Siebertz. Distributed dominating set approximations beyond planar graphs. *ACM Transactions on Algorithms (TALG)*, 15(3):1–18, 2019.

6 Marthe Bonamy, Linda Cook, Carla Groenland, and Alexandra Wesolek. A tight local algorithm for the minimum dominating set problem in outerplanar graphs, 2021. `arXiv:2108.02697`.

7 Andrzej Czygrinow, Michal Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *International Symposium on Distributed Computing*, DISC '08, pages 78–92. Springer, 2008.

8 Laurent Feuilloley. Bibliography of distributed approximation beyond bounded degree, 2020. `arXiv:2001.08510`.

9 Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., San Francisco, 1979.

10 Daniel Gonçalves. Edge partition of planar graphs into two outerplanar graphs. In *Proceedings of the thirty-seventh annual ACM Symposium on Theory of Computing*, STOC '05, pages 504–512, 2005.

11 Miikka Hilke, Christoph Lenzen, and Jukka Suomela. Brief announcement: local approximability of minimum dominating set on planar graphs. In *Proceedings of the 2014 ACM symposium on Principles of Distributed Computing*, PODC '14, pages 344–346, 2014.

12 Simeon Kublenz, Sebastian Siebertz, and Alexandre Vigny. Constant round distributed domination on graph classes with bounded expansion, 2021. `arXiv:2012.02701`.

13 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM (JACM)*, 63(2):1–44, 2016.

14 Christoph Lenzen, Yvonne Anne Oswald, and Roger Wattenhofer. What can be approximated locally? case study: Dominating sets in planar graphs. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 46–54, 2008.

15 Crispin St. John Alvah. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society (JLMS)*, s1-39(1):12–12, 1964.

16 Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability pcp characterization of np. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 475–484, 1997.

17 Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys (CSUR)*, 45(2):1–40, 2013.

18 Wojciech Wawrzyniak. Brief announcement: a local approximation algorithm for mds problem in anonymous planar networks. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 406–408, 2013.

19 Wojciech Wawrzyniak. A strengthened analysis of a local algorithm for the minimum dominating set problem in planar graphs. *Information Processing Letters (IPL)*, 114(3):94–98, 2014.

A Remaining casework

This appendix will provide the details of the case analysis from end of the proof of Lemma 2.

We are in the case in which $|\{v_1, v_2, w_1, w_2\}| \geq 3$. In particular, we may assume that s_1 has no neighbor in D.

▷ Claim 14. We have $v_1 \neq w_1$.

Proof. If not, then $v_1 = w_1$. By Claim 13, both v_2 and w_2 exist. Let G' be the outerplanar graph obtained from G by splitting the vertex v_1 into two vertices v_1' and w_1', both adjacent to s_1 and adjacent to each other, where v_1' is adjacent to $N[v_1] \cap N[s_2]$ and w_1' is adjacent to $N[v_1] \cap N[s_3]$. This gives three neighbors for both v_1' and w_1'. Since we can always add the edges $v_1'v_2$ and $w_1'w_2$ (which are chords of a cycle, using also that $N[s_1] \cap (N[s_2] \cup N[s_3]) = \emptyset$), we find $|B_S(G')| > |B_S(G)|$, whereas $D_S(G') = D_S(G)$, S is still dominating and G' is outerplanar. We find a contradiction with our assumption of the minimality of G. ◁

We henceforth assume that v_1, w_1 exist and are distinct and at least one of v_2, w_2 exists.

▷ Claim 15. The vertices v_2, w_2 exist.

Proof. By Claim 14 we can assume $v_1 \neq w_1$ and by Claim 12 we can assume v_2 exists. Suppose that w_2 does not exist. As in the previous case, $N[s_1]$ forms a vertex cut separating the component containing $N[s_2]$ from the component containing $N[s_3]$: if there was a path between $N[s_2]$ to $N[s_3]$ disjoint from $N[s_1]$, then we obtain a $K_{2,3}$-minor on vertex sets $N[s_2], \{s_1\}$ on one side and $\{v_1\}$, $\{v_2\}$, $N[s_3]$ on the other. When we delete $N[s_1]$, at most one of u_1, u_2 loses two neighbors, and the other (if it exists) loses only a single neighbor. The vertices x_1, x_2 can lose only a single neighbor. By Observation 10, after deleting $N[s_1]$ we can align the components of $N[s_2]$ and $N[s_3]$ in such a way that we can add the edges from $\{u_1x_1, u_2x_1, u_2x_2\}$. (For brevity, we handle the cases in which some of u_2, x_2 do not exist here as well, in which case we might add less edges.) We lost at most 4 vertices $B \cup D$, namely at most v_1, v_2, w_1 and one of the u_i (if one of them lost two neighbors). ◁

We have one final case in which v_1, v_2, w_1, w_2 all exist and are all distinct. Note that, as in Claim 15 above, $N[s_1]$ forms a vertex cut separating $N[s_2]$ from $N[s_3]$. We break this problem into three subcases: The case where x_2, u_2 both exists, the case where exactly one of x_2 and u_2 exists and the case where neither x_2 nor u_2 exists.

A.1 The case where x_2 and u_2 both exist

Suppose that x_2 and u_2 both exist. Note that at most one of u_1, u_2 and one of x_1, x_2 is adjacent to two vertices in $N[s_1]$. Let us assume without loss of generality that x_1, u_2 have at most one neighbor in $N[s_1]$. By Observation 10, after deleting $N[s_1]$, we can re-embed the graph in a way that we can add the edges u_1x_1, u_1x_2, u_2x_2. This gives a smaller counterexample, since we have "fixed" the degrees of u_1, u_2, x_1, x_2 and only lost $N[s_1] \cap (B \cup D)$, which has size at most 4.

A.2 The case when exactly one of x_2, u_2 exists

Suppose now that only u_2 exists. (The case in which only x_2 exists is analogous.) Note that s_2 can have at most one neighbor in D. See also Figure 11.

- Suppose first that s_2 has a neighbor $d \in D$. We delete and add edges (if needed) and renumber such that u_1 is adjacent to v_1, u_2 to v_2 and u_1 to u_2, but no other edges among $\{u_1, u_2, v_1, v_2\}$ are present. Now we can add the chords u_1s_1, u_2s_1 to the cycle $s_1v_1u_1s_2u_2v_2s_1$. We delete s_2 and d. We have now lost at most four vertices from $B \cup D$: namely at most v_1, v_2, d and one of the u_i (if it was adjacent to v_1 and v_2 originally).

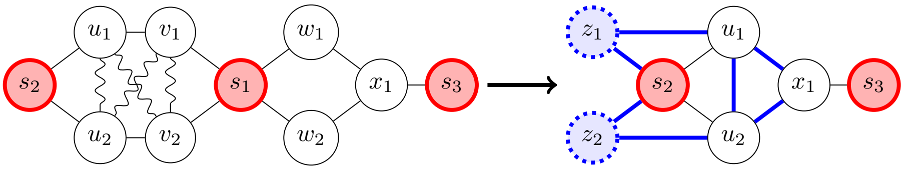


Figure 11 An illustration of the case v_1, v_2, w_1, w_2, u_2 all exist and are all distinct and x_2 does not exist. At top we show the case where s_2 has a neighbor $d \in D$. Some of the wavy edges may be present in G. As usual, there may be other edges present in G that have not been drawn, but they are not relevant to our argument. At the bottom we illustrate the case where s_2 has no neighbor in D. For $i \in \{1, 2\}$, we add the vertices z_i and edges z_is_2, z_iu_i if needed to make $\deg(u_i) \geq 4$ in G'.

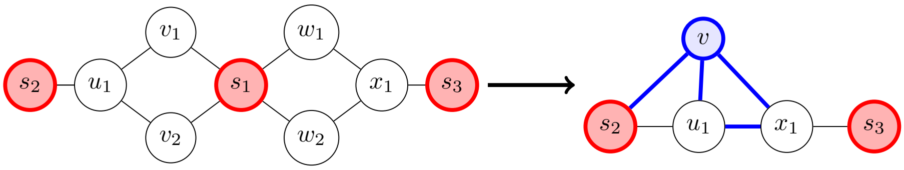


Figure 12 An illustration of the reduction for the case where u_2, w_2 do not exist and s_2 has no neighbors in D. We only draw edges that are relevant to our argument.

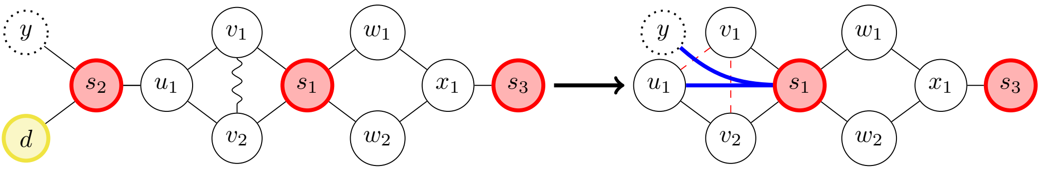


Figure 13 An illustration of the reduction for the case where u_2, w_2 do not exist and s_2 has exactly one neighbor $d \in D$. We use dashed red lines to emphasize that v_1 is not adjacent to u_1, u_2 in G' and a wavy line to show that v_1 may be adjacent to v_2 in G. As usual, we do not draw edges that are not relevant to our argument.

- Suppose now that s_2 has no neighbors in D. We remove $N[s_1]$ and add the edges u_1x_1,u_2x_1 and u_1u_2 if it is not already present. If u_1 has degree 3, then it has no neighbors outside of u_2, x_1, s_2 and so we may create a new vertex adjacent to both u_1 and s_2. Similarly, we can fix the degree of u_2 if needed. Note that since s_1 has degree at least four, $w_1, w_2 \notin D$. If we lost one of w_1, w_2 from B, then $x_1 \notin D$. In that case we lose at most v_1, v_2, w_1, w_2 from $B \cup D$.If $x_1 \in D$ we lose at most v_1, v_2, x_1 from $B \cup D$. In both cases we found a smaller counterexample.

A.3 The case when neither u_2 nor x_2 exists

If both u_1 and x_1 do not have degree exactly four, then we can remove $N[s_1]$ and add the edge u_1x_1; in this case we only lose a subset of $\{v_1, v_2, w_1, w_2\}$ from $B \cup D$. Hence we can assume by symmetry that u_1 has degree exactly four.

- We first handle the case in which s_2 has no neighbor in D. Since u_1 has degree exactly 4, after removing $N[s_1]$ we can create a new vertex v and add the edges u_1v, s_2v, x_1v, u_1x_1. As a result, we have lost at most v_1, v_2, w_1, w_2 from $B \cup D$ and found a smaller counterexample. See Figure 12.
- Suppose now that s_2 has only neighbors in $D \cup \{u_1\}$, which we name d_1, d_2 (where d_2 may or may not exist). We remove $N[s_2] \setminus \{u_1\}$ (at most three vertices), remove the edge v_1v_2 and add the edge u_1s_1. We again found a smaller counterexample as the only vertices we may have lost from $B \cup D$ are v_1, v_2, d_1, d_2.
- Suppose now that s_2 has exactly one neighbor $d \in D$. It may have another neighbor $y \neq u_1, d$, which if it exists, is not in D. We delete the vertices s_2, d as well as the edges u_1v_1 and v_1v_2 (if these exist). As u_1 was a cut-vertex previously, we can now add the edges u_1s_1 and ys_1 (say along the path $u_1v_2s_1$) to ensure that the size of the dominating set has dropped by one whereas we lost at most d, u_1, v_1, v_2 from $B \cup D$. See Figure 13.

Fast Nonblocking Persistence for Concurrent Data Structures

Wentao Cai[1]
University of Rochester, NY, USA

Haosen Wen[1]
University of Rochester, NY, USA

Vladimir Maksimovski
University of Rochester, NY, USA

Mingzhe Du
University of Rochester, NY, USA

Rafaello Sanna
University of Rochester, NY, USA

Shreif Abdallah
University of Rochester, NY, USA

Michael L. Scott
University of Rochester, NY, USA

Abstract

We present a fully lock-free variant of our recent Montage system for persistent data structures. The variant, nbMontage, adds persistence to almost any nonblocking concurrent structure without introducing significant overhead or blocking of any kind. Like its predecessor, nbMontage is *buffered durably linearizable*: it guarantees that the state recovered in the wake of a crash will represent a consistent prefix of pre-crash execution. Unlike its predecessor, nbMontage ensures wait-free progress of the persistence frontier, thereby bounding the number of recent updates that may be lost on a crash, and allowing a thread to force an update of the frontier (i.e., to perform a `sync` operation) without the risk of blocking. As an extra benefit, the helping mechanism employed by our wait-free `sync` significantly reduces its latency.

Performance results for nonblocking queues, skip lists, trees, and hash tables rival custom data structures in the literature – dramatically faster than achieved with prior general-purpose systems, and generally within 50% of equivalent non-persistent structures placed in DRAM.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms; Computer systems organization → Reliability; Theory of computation → Parallel computing models

Keywords and phrases Persistent Memory, Nonblocking Progress, Buffered Durable Linearizability

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.14

Related Version *Full Version*: `https://arxiv.org/abs/2105.09508`

Supplementary Material *Software (Source Code)*: `https://github.com/urcs-sync/Montage`
archived at `swh:1:dir:421cef8ca9c97f07099163c791aa8316c60fda86`

Funding This work was supported in part by NSF grants CCF-1717712, CNS-1900803, CNS-1955498, and by a Google Faculty Research award.

1 Introduction

With the advent of dense, inexpensive nonvolatile memory (NVM), it is now feasible to retain pointer-based, in-memory data structures across program runs and even hardware reboots. So long as caches remain transient, however, programs must be instrumented with write-back and fence instructions to ensure that such structures remain consistent in the wake of a crash. Minimizing the cost of this instrumentation remains a significant challenge.

[1] The first two authors (Wentao Cai and Haosen Wen) contributed equally to this paper.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 14; pp. 14:1–14:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Section 2 of this paper surveys both bespoke persistent data structures and general-purpose systems. Most of these structures and systems ensure that execution is *durably linearizable* [28]: not only will every operation linearize (appear to occur atomically) sometime between its call and return; each will also *persist* (reach a state in which it will survive a crash) before returning, and the persistence order will match the linearization order. To ensure this agreement between linearization and persistence, most existing work relies on locks to update the data structure and an undo or redo log together, atomically.

Unfortunately, strict durable linearizability forces expensive write-back and fence instructions onto the critical path of every operation. A few data structures – e.g., the Dalí hash table of Nawab et al. [43] and the InCLL MassTree of Cohen et al. [10] – reduce the cost of instrumentation by supporting only a relaxed, *buffered* variant [28] of durable linearizability. Nawab et al. dub the implementation technique *periodic persistence.* In the wake of post-crash recovery, a buffered durably linearizable structure is guaranteed to reflect some prefix of the pre-crash linearization order – but with the possibility that recent updates may be lost.

Independently, some data structures – notably, the B-tree variants of Yang et al. [54], Oukid et al. [44], and Chen et al. [7], and the hash set of Zuriel et al. [55] – reduce the overhead of persistence by writing back and fencing only the data required to rebuild a semantically equivalent structure during post-crash recovery. A mapping, for example, need only persist a pile of key-value pairs. Index structures, which serve only to increase performance, can be kept in faster, volatile memory.

Inspired by Dalí and the InCLL MassTree, our group introduced the first *general purpose* system [53] for buffered durably linearizable data structures. Like the sets of the previous paragraph, *Montage* facilitates persisting only semantically essential *payloads.* Specifically, it employs a global *epoch clock.* It tracks the semantically essential updates performed in each operation and ensures that (1) no operation appears to span an epoch boundary, and (2) no essential update fails to persist for two consecutive epochs. If a crash occurs in epoch e, Montage recovers the abstract state of the structure from the end of epoch $e - 2$.

In return for allowing a bit of work to be lost on a crash, and for being willing to rebuild, during recovery, any portion of a structure needed only for good performance, users of Montage can expect to achieve performance limited primarily by the read and write latency of the underlying memory – not by instrumentation overhead. When a strict persistence guarantee is required (e.g., before printing a completion notice on the screen or responding to a client over the network), Montage allows an application thread to invoke an explicit `sync` operation – just as it would for data in any conventional file or database system.

Unfortunately, like most general-purpose persistence systems, Montage relies on locks at critical points in the code. All of the reported performance results are for lock-based data structures. More significantly, while the system can support nonblocking structures, progress of the persistence frontier is fundamentally blocking. Specifically, before advancing the epoch from e to $e + 1$, the system waits for all operations active in epoch $e - 1$ to complete. If one of those operations is stalled (e.g., due to preemption), epoch advance can be indefinitely delayed. This fact implies that the `sync` operation cannot be nonblocking. It also implies that the size of the history suffix that may be lost on a crash cannot be bounded *a priori.*

While blocking is acceptable in many circumstances, nonblocking structures have compelling advantages. In the original paper on durable linearizability, Izraelevitz et al. [28] presented a mechanical transform that will turn any linearizable nonblocking concurrent data structure into an equivalent persistent structure. Unfortunately, this transform is quite expensive, especially for highly concurrent programs: it litters the code with write-back and fence instructions. More efficient strategies for several specific data structures, including queues [19] and hash tables [8, 55], have also been developed by hand.

Friedman et al. [18] observed that many nonblocking operations begin with a read-only "traversal" phase in which instrumentation can, with care, be safely elided; applying this observation to the transform of Izraelevitz et al. leads to substantially better performance in many cases, but the coding process is no longer entirely mechanical. Ramalhete et al. [46] and Beadle et al. [1] present nonblocking persistent software transactional memory (STM) systems, but both have fundamental serial bottlenecks that limit scalability.

In this paper, we extend Montage to produce the first *general-purpose, high-performance system for nonblocking periodic persistence.* Our *nbMontage* allows programmers, in a straightforward way, to convert most wait-free or lock-free linearizable concurrent data structures into fast, equivalent structures that are lock-free and buffered durably linearizable. (Obstruction-free structures can also be converted; they remain obstruction free.) Like its predecessor, nbMontage requires every nonblocking update operation to linearize at a compare-and-swap (CAS) instruction that is guaranteed, prior to its execution, to constitute the linearization point if it succeeds. (Read-only operations may linearize at a `load`.) Most nonblocking structures in the literature meet these requirements.

Unlike its predecessor, nbMontage provides a wait-free `sync`. Where the original system required all operations in epoch $e - 1$ to complete before the epoch could advance to $e + 1$, nbMontage allows pending operations to remain in limbo: in the wake of a crash or of continued crash-free execution, an operation still pending at the end of $e - 1$ may or may not be seen, sometime later, to have linearized in that epoch. In addition, where the original Montage required operations to accommodate "spurious" CAS failures caused by epoch advance, nbMontage retries such CASes internally, without compromising lock freedom. These changes were highly nontrivial: they required new mechanisms to register (announce) pending updates; distinguish, in recovery, between registered and linearized updates; indefinitely delay (and reason about) the point at which an update is known to have linearized; and avoid work at `sync` time for threads that have nothing to persist.

We have applied nbMontage to Michael & Scott's queue [41], Natarajan & Mittal's binary search tree [42], the rotating skip list of Dick et al. [15], Michael's hash table [39], and Shalev & Shavit's extensible hash table [48]. All conversions were straightforward – typically less than 30 lines of code. By persisting only essential data and avoiding writes-back and fences on the critical path, nbMontage structures often approach or outperform not only their equivalents in the original (blocking) Montage but also their transient equivalents when data is placed (without any algorithmic changes) in slower, nonvolatile memory.

Summarizing contributions:

1. We introduce nbMontage, the first general system for nonblocking periodic persistence.
2. We tailor the nbMontage API to nonblocking data structures. With this new API, conversion of existing nonblocking structures for persistence is straightforward.
3. We argue that nbMontage provides buffered durable linearizability and wait-free `sync` for compatible data structures, while still preserving safety and liveness.
4. We compare the performance of nbMontage structures both to their original, transient versions and to hand-crafted alternatives from the literature, running on a recent Intel server with Optane NVM. Our results confirm exceptional throughput, responsive `sync`, negligible overhead relative to the original Montage, and reasonable recovery latency.

2 Related Work

The past decade has seen extensive work on persistent data structures, much of it focused on B-tree indices for file systems and databases [3, 6, 7, 26, 29, 31, 44, 50, 54]. Other work has targeted queues [19], RB trees [52], radix trees [30], and hash tables [8, 43, 47, 55].

In recent years, *durable linearizability* has emerged as the standard correctness criterion for such structures [19, 28, 37, 55]. This criterion builds on the familiar notion of linearizability [24] for concurrent (non-persistent) structures. A structure is said to be linearizable if, whenever threads perform operations concurrently, the effect is as if each operation took place, atomically, at some point between its call and return, yielding a history consistent with the sequential semantics of the abstraction represented by the structure. A persistent structure is said to be *durably linearizable* if (1) it is linearizable during crash-free execution, (2) each operation persists (reaches a state that will survive a crash) at some point between its call and return, and (3) the order of persists matches the linearization order.

In addition to custom data structures, several groups have developed general-purpose systems to ensure the failure atomicity of lock-based critical sections [4, 25, 27, 33] or speculative transactions [1, 5, 9, 11, 13, 20–22, 32, 38, 45, 46, 49, 51]. Like the bespoke structures mentioned above, all of these systems are durably linearizable – they ensure that an operation has persisted before returning to the calling thread.

As noted in Section 1, nonblocking persistent structures can achieve failure atomicity without the need for logging, since every reachable concrete state corresponds to a well-defined abstract state. At the same time, while a lock-based operation can easily arrange to linearize and persist in the same order (simply by holding onto the locks needed by any successor operation), a nonblocking operation becomes visible to other threads the moment it linearizes. As a result, those other threads must generally take care to ensure that anything they read has persisted before they act upon it. Writing back and fencing read locations is a major source of overhead in the mechanical transform of Izraelevitz et al. [28]. Friedman et al. [18] avoid this overhead during the initial "traversal" phase of certain nonblocking algorithms. David et al. [14] avoid redundant writes-back and fences in linked data structures by *marking* each updated pointer in one of its low-order bits. A thread that persists the pointer can use a CAS to clear the bit, informing other threads that they no longer need to do so.

In both blocking and nonblocking structures, the overhead of persistence can be reduced by observing that not all data are semantically meaningful. In any implementation of a set or mapping, for example, the items or key-value pairs must be persistent, but the index structure can (with some effort) be *rebuilt* during recovery. Several groups have designed persistent B-trees, lists, or hash tables based on this observation [7, 35, 44, 54].

Ultimately, however, any data structure or general-purpose system meeting the strict definition of durable linearizability will inevitably incur the overhead of at least one persistence fence in every operation [12]. For highly concurrent persistent structures, this overhead can easily double the latency of every operation. Similar observations, of course, have applied to I/O operations since the dawn of electronic computing, and are routinely addressed by completing I/O in the background. For data structures held in NVM, *periodic persistence* [43] provides an analogous "backgrounding" strategy, allowing a structure to meet the more relaxed requirements of *buffered* durable linearizability – specifically, the state recovered after a crash is guaranteed to represent a prefix of the linearization order of pre-crash execution.

Nawab et al. [43] present a lock-based hash table, Dalí, that performs updates only by pre-pending to per-bucket lists, thereby creating a revision history (deletions are effected by inserting "anti-nodes"). A clever "pointer-rotation" strategy records, for each bucket, the head nodes of the list for each of the past few values of a global *epoch clock*. At the end of each coarse-grain epoch, the entire cache is flushed. In the wake of a crash, threads ignore nodes prepended to hash bucket lists during the two most recent epochs. No other writes-back or fences are required. Cohen et al. [10] also flush the cache at global epoch boundaries, but employ a clever system of *in-cache-line-logging* (InCLL) to retain the epoch number and beginning-of-epoch value for every modified leaf-level pointer in a variant of the Masstree structure [36]. In the wake of a crash, threads use the beginning-of-epoch value for any pointer that was modified in the epoch of the crash.

Both Dalí and InCLL employ techniques that might be applicable in principle to other data structures. To the best of our knowledge, however, Montage [53] is the only existing general-purpose system for buffered durable linearizable structures. It also has the advantage of persisting only semantically essential data. As presented, unfortunately, it provides only limited support for nonblocking data structures, and its attempts to advance the epoch clock can be arbitrarily delayed by stalled worker threads. Our nbMontage resolves these limitations, allowing us to provide a wait-free `sync` operation and to bound the amount of work that may be lost on a crash. It also provides a substantially simpler API.

3 System Design

3.1 The Original Montage

Semantically essential data in Montage resides in *payload* blocks in NVM. Other data may reside in transient memory. The original system API [53] is designed primarily for lock-based data structures, but also includes support for nonblocking operations (with blocking advance of the persistence frontier). The typical operation is bracketed by calls to `begin_op` and `end_op`. In between, reads and writes of payloads use special *accessor* (`get` and `set`) methods.

Internally, Montage maintains a slow-ticking *epoch clock*. In the wake of a crash in epoch e, the Montage recovery procedure identifies all and only the payloads in existence at the end of epoch $e-2$. It provides these, through a parallel iterator, to the restarted application, which can then rebuild any needed transient structures. Accessor methods allow payloads that were created in the current epoch to be modified in place, but they introduce significant complexity to the programming model.

Payloads are created and deleted with `pnew` and `pdelete`. These routines are built on a modified version of our Ralloc [2] persistent allocator. In the original Ralloc, a tracing garbage collector was used in the wake of a crash to identify and reclaim free blocks. In the Montage version, epoch *tags* in payloads are used to identify all and only those blocks created and not subsequently deleted at least two epochs in the past. To allow a payload to survive if a crash happens less than two epochs after a deletion, deletions are implemented by creating *anti-nodes*. These are automatically reclaimed, along with their corresponding payloads, after two epochs have passed.

To support nonblocking operations, the original Montage provides a `CAS_verify` routine that succeeds only if it can do so in the same epoch as the preceding `begin_op`. If `CAS_verify` fails, the operation will typically start over; before doing so, it should call `abort_op` instead of `end_op`, allowing the system to clean up without persisting the operation's updates.

Regardless of persistence, nodes removed from a nonblocking structure typically require some sort of *safe memory reclamation* (SMR) – e.g., epoch-based reclamation [17] or hazard pointers [40] – to delay the reuse of memory until one can be sure that no thread still holds a transient reference. In support of SMR, the original Montage provides a `pretire` routine that creates an anti-node to mark a payload as semantically deleted, and no new operation can reference this payload. In the absence of crashes, Montage will automatically reclaim the payload and its anti-node 2–3 epochs after SMR calls `pdelete`. In the event of a crash, however, if two epochs have elapsed since `pretire`, the existence of the anti-node allows the Montage recovery procedure to avoid a memory leak by reclaiming the retired payload even in the absence of `pdelete`. This is safe since the epoch of the `pretire` is persisted, and all

```
class PBlk; // Base class of all Payload classes
class Recoverable { // Base class of all persistent objects
  template <class payload_type> payload_type* pnew(...); // Create a payload block
  void pdelete(PBlk*); // Delete a payload; should be called only when safe, e.g., by SMR.
  void pdetach(PBlk*); // Mark payload for retirement if operation succeeds
  void sync(); // Persist all operations that happened before this call
  vector<PBlk*>* recover(); // Recover and return all recovered payloads
  void abort_op(); // Optional routine to abandon current operation
};
template <class T> class CASObj { // Atomic CAS object that provides load and CAS
  /* Epoch-verifying linearization method: */
  bool lin_CAS(T expected, T desired) {
    begin_op(); // write back or delete old payloads as necessary; tag new ones
    while (1) { // iterations can be limited for liveness
      ... // main body of DCSS
      switch (DCSS_status) {
        case COMMITTED: end_op(); return true; // clean up metadata
        case FAILED: abort_op(); return false; // untag payloads and clear retirements; clean up metadata
        case EPOCH_ADVANCED: reset_op(); // update and reuse payloads and retirements
      }
    }
  /* Non-verifying atomic methods: */
  T load(); void store(T desired); bool CAS(T expected, T desired);
};
```

Figure 1 C++ API of nbMontage, largely inherited from the original Montage.

operations with references to this payload are gone after the crash. Significantly, since a still-active operation will (in the original Montage) prevent the epoch from advancing far enough to persist anything in its epoch, `pretire` can safely be called after the operation has linearized, so long as it has not yet called `end_op`.

When a program needs to ensure that operation(s) have persisted (e.g., before printing a confirmation on the screen or responding to a client over the network), Montage allows it to invoke an explicit `sync`. The implementation simply advances the epoch from e to $e + 2$ (waiting for operations in epochs $e - 1$ and e to complete). The two-epoch convention avoids the need for quiescence [43]: a thread can advance the epoch from e to $e + 1$ while other threads are still completing operations that will linearize in e.

The key to buffered durable linearizability is to ensure that every operation that updates payloads linearizes in the epoch with which those payloads are tagged. Each epoch boundary then captures a prefix of the data structure's linearization order. Maintaining this consistency is straightforward for lock-based operations. In the nonblocking case, `CAS_verify` uses a variant of the double-compare-single-swap (DCSS) construction of Harris al. [23] (see App. B in Full Version for complete pseudocode) to confirm the expected epoch and perform a conditional update, atomically. Unfortunately, the fact that an epoch advance from e to $e+1$ must wait for operations in $e - 1$ means that even if a data structure remains nonblocking during crash-free execution, the progress of persistence itself is fundamentally blocking. This in turn implies that calls to `sync` are blocking, and that it is not possible, *a priori*, to bound the amount of work that may be lost on a crash.

Note, however, that since `CAS_verify` will succeed only in the expected epoch, any nonblocking operation that lags behind an epoch advance is doomed to fail and retry in a subsequent epoch. There is no need to wait for it to resume, explicitly fail, and unregister from the old epoch in `abort_op`. Unfortunately, the waiting mechanism is deeply embedded in the original Montage implementation – e.g., in the implementation of `pretire`, as noted above. Overall, there are four nontrivial issues that must be addressed to build nbMontage:

(Sec. 3.3) Every operation must register its pending updates (both new and to-be-deleted payloads) before reaching its linearization point, so that an epoch advance can help it to persist even if it stalls immediately after the linearization point.

(Sec. 3.4) The recovery procedure must be able to distinguish an epoch's "real" payloads and anti-nodes from those that may have been registered for an operation that failed due to a CAS conflict or epoch advance.

(Sec. 3.5) The buffering containers that record persistent blocks to be written back or deleted need a redesign, in order to accommodate an arbitrary number of epochs in which operations have not yet noticed that they are doomed to fail and retry in a new epoch.

(Sec. 3.6) An epoch advance must be able to find and persist any blocks (payloads or anti-nodes) that were created in the previous epoch but have not yet been written back and fenced. If `sync` is to be fast, this search must avoid iterating over all active threads.

3.2 nbMontage API

As shown in Figure 1, the nbMontage API reflects three major changes. First, because the epoch can now advance from e to $e+1$ even when an operation is still active in $e-1$, we must consider the possibility that a thread may remove a payload from the structure, linearize, and then stall. If a crash occurs two epochs later, we must ensure that the removed payload is deleted during post-crash recovery, to avoid a memory leak: post-linearization `pretire` no longer suffices. Our nbMontage therefore replaces `pretire` with a `pdetach` routine that must be used to register to-be-deleted payloads *prior* to linearization. As in the original Montage, deletion during crash-free execution is the responsibility of the application's SMR.

Second, again because of nonblocking epoch advance, nbMontage requires that payloads visible to more than one thread be treated as immutable. Updates always entail the creation of new payloads; `get` and `set` accessors are eliminated.

Third, in a dramatic simplification, nbMontage replaces the original `begin_op`, `end_op`, and `CAS_verify` with a new `lin_CAS` (linearizing CAS) routine. (The `abort_op` routine is also rolled into `lin_CAS`, but remains in the API so operations can call it explicitly if they choose, for logical reasons, to start over.) When `lin_CAS` is called, all payloads created by the calling thread since its last completed operation (and not subsequently deleted) will be tagged with the current epoch, e. All anti-nodes stemming from `pdetach` calls made since the last completed operation (and not corresponding to payloads created in that interval) will likewise be tagged with e. The `lin_CAS` will then attempt a DCSS and, if the current epoch is still e and the specified location contains the expected value, the operation will linearize, perform the internal cleanup previously associated with `end_op`, and return `true`. If the DCSS fails because of a conflicting update, `lin_CAS` will perform the internal cleanup associated with `abort_op` and return `false`. If the DCSS fails due to epoch advance, `lin_CAS` will update the operation's payloads and anti-nodes to the new epoch and retry. By ensuring, internally, that the epoch never advances unless some thread has completed an operation (App. C in Full Version), nbMontage can ensure that some thread makes progress in each iteration of the retry loop.

Programmers using nbMontage are expected to obey the following constraints:

1. Each nbMontage data structure R must be designed to be nonblocking and linearizable during crash-free execution when nbMontage is disabled. Specifically, R must be linearizable when (a) `pnew` and `pdelete` are ordinary `new` and `delete`; (b) `pdetach` and `sync` are no-ops; and (c) `CASObj` is `std::atomic` and `lin_CAS` is ordinary `CAS`.
2. Every update operation of R linearizes in a pre-identified `lin_CAS`– one that is guaranteed, before the call, to comprise the operation's linearization point if it succeeds. Any operation that conflicts with or depends upon this update must access the `lin_CAS`'s target location. Read-only operations may linearize at a `load`.

3. Once attached to a structure (made visible to other threads), a payload is immutable. Changes to a structure are made only by adding and removing payloads.
4. The semantics of each operation are fully determined by the set of payloads returned by `pnew` and/or passed to `pdetach` prior to `lin_CAS`.

Pseudocode for nbMontage's core classes and methods appears in Figure 2; these are discussed and referred to by pseudocode line numbers in the following subsections. Appendix A presents the changes required to port Maged Michael's lock-free hash table [39] to nbMontage.

3.3 Updates to Payloads

To allow the epoch clock to advance without blocking, nbMontage abandons in-place update of payloads. It interprets `pdetach` as requesting the creation of an *anti-node*. An anti-node shares an otherwise unique, hidden ID with the payload being detached. Newly created payloads and anti-nodes are buffered until the next `lin_CAS` in their thread. If the `lin_CAS` succeeds, the buffered nodes will be visible to epoch advance operations, and will persist even if the creating thread has stalled.

In the pseudocode of Figure 2, calls to `pnew` and `pdetach` are held in the `allocs` and `detaches` containers. Anti-nodes are created, and both payloads and anti-nodes are tagged, in `begin_op` (lines 69–76, called from within `lin_CAS`). If the `lin_CAS` fails due to conflict, `abort_op` resets `pnew`-ed payloads so they can be reused in the operation's next attempt (lines 91–92); it also withdraws `pdetach` requests, allowing the application to detach something different the next time around (lines 87–90). If attempts occur in a loop (as is common), the programmer may call `pnew` outside the loop and `pdetach` inside, as shown in Figure 7 (App. A). If an operation no longer needs its `pnew`-ed payloads (e.g., after a failed insertion), it may call `pdelete` to delete them; this automatically erases them from `allocs` (line 56). The internal `reset_op` routine serves to update and reuse both payloads and anti-nodes in anticipation of retrying a DCSS that fails due to epoch advance (lines 95–97).

3.4 CAS and Recovery

The implementation of `lin_CAS` employs an array of persistent *descriptors*, one per thread. These form the basis of the DCSS construction [23] (App. B in Full Version). Each descriptor records CAS parameters (the old value, new value, and CAS object); the epoch in which to linearize; and the status of the CAS itself (in progress, committed, or failed – lines 29–37). After a crash, the recovery procedure must be able to tell when a block in NVM (a payload or anti-node) appears to be old enough to persist, but corresponds to an operation that did not commit. Toward that end, each block contains a 64-bit field that encodes the thread ID and a monotonic serial number; together, these constitute a unique *operation ID* (lines 32 and 43). At the beginning of each operation attempt, `begin_op` updates the descriptor, incrementing its serial number (line 65). Previous uses of the descriptor with smaller serial numbers are regarded as having committed; blocks corresponding to those versions remain valid unless they are deleted or reinitialized (lines 67, 88, and 92). Deleting or reinitializing a persistent block resets its epoch to zero and registers it to be written back in the current epoch (lines 44–51). Registration ensures that resets persist, in `begin_op`, *before the next update to the descriptor* (lines 61–65). During an epoch advance from e to $e + 1$, the descriptors of operations in $e - 1$ are written back (at line 102) to ensure that their statuses reach NVM before the update of the global epoch clock.

Informally, an nbMontage payload is said to be *in use* if it has been created and not yet detached by linearized operations. Identifying such payloads precisely is made difficult by the existence of *pending* operations – those that have started but not yet completed, and

```
1   Struct CircBuffer
2       uint64 cap
3       atomic<uint64> pushed,popped
4       PBlk* blks[cap]
5       Function push(PBlk* item)
6           cpush=pushed.load()
7           cpop=popped.load()
8           if cpush≥cpop+cap then
9               clwb(blks[cpop%cap])
10              popped.CAS(cpop,cpop+1)
11          blks[cpush%cap]=item
12          pushed.store(cpush+1) // single producer
13      Function pop_all()
14          cpop=popped.load()
15          cpush=pushed.load()
16          if cpop==cpush then
17              return
18          foreach i from cpop to cpush do
19              break if i%cap reaches cpop%cap twice
20              clwb(blks[i%cap])
21          popped.CAS(cpop,cpush)
22  atomic<uint64> global_epoch
23  Mindicator mindi
24  thread_local uint64 e_curr,e_last
25  thread_local vector<PBlk*> allocs,detaches
26  thread_local int tid
27  int thd_cnt // number of threads
28  CircBuffer TBP[thd_cnt][4]
29  Struct Desc
30      uint64 old,new,epoch=0
31      uint64 type=DESC
32      uint64 tid_sn=0
        // 64 bits for ref to CASObj
        // 64 for cnt, with last 2 for status
33      atomic<uint128> r_c_s
34      Function reinit(uint64 e)
35          tid_sn++
36          r_c_s.store(0)
37          epoch=e
38  Struct PBlk
39      (void* vtable)
40      PBlk* anti=NULL
41      uint64 epoch=0
42      uint64 type={PAYLOAD,ANTI}
43      uint64 tid_sn=0,blk_uid=0
44      Function setup(uint64 t,Desc* desc,uint64 e)
45          type=t
46          epoch=desc?0:desc.epoch
47          tid_sn=desc?0:desc.tid_sn
48          TBP[tid][e%4].push(this)
49      Function destructor()
50          epoch=0
51          clwb(this)
52  Desc descs[thd_cnt]
53  vector<PBlk*> TBF[thd_cnt][4]
54  Function pdelete(PBlk* pblk)
55      e=global_epoch.load()
56      allocs.erase(pblk) // no-op if pblk not in allocs
57      TBF[tid][(e+1)%4].insert(pblk→anti)
58      TBF[tid][e%4].insert(pblk)
59  Function begin_op(bool reset=false)
60      e_curr=global_epoch.load()
61      if e_last<e_curr then
62          TBP[tid][e_last%4].pop_all()
63          clwb(descs[t])
64          update t's val to e_curr in mindi
65      descs[tid].reinit(e_curr)
66      for i from e_last-1 to min(e_last+1,e_curr-2) do
67          delete all items in TBF[tid][i%4]
68          sfence
69      foreach r in detaches do
70          if !reset then // default branch
71              allocate an anti-node anti for r
72              r→anti=anti
73              r→anti→blk_uid=r→blk_uid
74          r→anti→setup(ANTI,descs[tid],e_curr)
75      foreach p in allocs do
76          p→setup(PAYLOAD,descs[tid],e_curr)
77  Function end_op()
78      detaches.clear()
79      allocs.clear()
80      e_last=e_curr
81      e_curr=0
82  Function abort_op(bool reset=false)
83      if reset then // to reuse anti-nodes
84          foreach r in detaches do
85              r→anti→setup(ANTI,NULL,e_curr)
86      else // default branch:delete anti-nodes
87          foreach r in detaches do
88              delete(r→anti)
89              r→anti=NULL
90          detaches.clear()
91      foreach p in allocs do
92          p→setup(PAYLOAD,NULL,e_curr)
93      e_last=e_curr
94      e_curr=0
95  Function reset_op()
96      abort_op(true)
97      begin_op(true)
98  Function advance()
99      e=global_epoch.load()
100     foreach t in mindi whose val==e-1 do
101         TBP[t][(e-1)%4].pop_all()
102         clwb(descs[t])
103         update t's val to e in mindi
104     sfence
105     if some op linearized in e-1 or e then
106         global_epoch.CAS(e,e+1)
```

Figure 2 nbMontage pseudocode.

whose effects may not yet have been seen by other threads. In the wake of a crash in epoch e, nbMontage runs through the Ralloc heap, assembling a set of potentially allocated blocks and finding all CAS descriptors (identified by their `type` fields – lines 31 and 42). By matching the serial numbers and thread IDs of blocks and descriptors, the nbMontage-internal recovery procedure identifies all and only the payloads that are known, as of the crash, to have been in use at the end of epoch $e-2$. Specifically, if block B has thread ID t, serial number s, and epoch tag f, nbMontage will recover B if and only if

1. $0 < f \leq e-2$;
2. $(s < \texttt{descs}[t]\texttt{.sn}) \vee (s = \texttt{descs}[t]\texttt{.sn} \wedge \texttt{descs}[t]\texttt{.status} = \texttt{COMMITTED})$; and
3. if B is a payload, it has not been canceled by an in-use anti-node.

Once the in-use blocks have been identified, nbMontage returns them to a data-structure-specific recovery routine that rebuilds any needed transient state, after which the state of the structure is guaranteed to reflect some valid linearization of pre-crash execution through the end of epoch $e - 2$.

3.5 Buffering Containers

Persistent blocks created or deleted in a given epoch will be recorded in thread- and epoch-specific to-be-persisted (TBP) and to-be-freed (TBF) containers. Every thread maintains four statically allocated instances of each (only 3 are actually needed, but indexing is faster with 4 – Fig. 2, lines 28 and 53).

TBPs are fixed-size circular buffers. When a buffer is full, its thread removes and writes back a block before inserting a new one. In the original version of Montage, epoch advance always occurs in a dedicated background thread (the `sync` operation handshakes with this thread). As part of the advance from epoch e to $e + 1$, the background thread iterates over all worker threads t, waits for t to finish any active operation in $e - 1$, extracts all blocks from $\text{TBP}[t][(e - 1) \bmod 4]$, and writes those blocks back to memory.

Insertions and removals from a TBP buffer never occur concurrently in the original version of Montage. In nbMontage, however, an operation that is lagging behind may not yet realize that it is doomed to retry, and may still be inserting items into the buffer when another thread (or several other threads!) decide to advance the epoch. The lagging thread, moreover, may even be active in epoch $e - 1 - 4k$, for some $k > 0$ (lines 88 and 92). This concurrency implies that TBPs need to support single-producer-multiple-consumers (SPMC) concurrency. Our implementation of the SPMC buffer (lines 1–21) maintains two monotonic counters, `pushed` and `popped`. To insert an item, a thread uses `store` to increment `pushed`. To remove some item(s), a thread uses `CAS` to increase `popped`. For simplicity, the code exploits the fact that duplicate writes-back are semantically harmless: concurrent removing threads may iterate over overlapping ranges (lines 13–21).

TBFs are dynamic-size, thread-unsafe containers implemented as vectors (line 53). Although deletion must respect epoch ordering, it can be performed lazily during crash-free execution, with each thread responsible for the blocks in its own TBFs. In `begin_op`, *after it has updated its descriptor*, thread t deletes blocks in $\text{TBF}[t][i \bmod 4]$, for $i \in [e_{last} - 1, \min(e_{last} + 1, e_{curr} - 2)]$, where e_{last} is the epoch of t's last operation and e_{curr} is the epoch of its current operation (lines 66–68).

3.6 Epoch Advance

To make `sync` nonblocking, we first decentralize the original epoch advance in Montage so that instead of making a request of some dedicated thread, every thread is now able to advance the epoch on its own. In the worst case, such an epoch advance may require iterating over the TBP buffers of all threads in the system. In typical cases, however, many of those buffers may be empty. To reduce overhead in the average case, we deploy a variant of Liu et al.'s *mindicator* [34] to track the oldest epoch in which any thread may still have an active operation. Implemented as a wait-free, fixed-size balanced tree, our variant represents each thread and its current epoch as a leaf. An ancestor in the tree indicates the minimum epoch of all nodes in its subtree. When thread t wishes to advance the epoch from e to $e + 1$, it first checks to see whether the root of the mindicator is less than e. If so, it scans up the tree from its own leaf until it finds an ancestor with epoch $< e$. It then reverses course, traces down the tree to find a lagging thread, persists its descriptor and any blocks in the requisite

TBP container, and repeats until the root is at least e. When multiple threads call `sync` concurrently, this nearest-common-ancestor strategy allows the work of persistence to be effectively parallelized. Experiments described in Section 5.3 confirm that our use of the mindicator, together with the lazy handling of TBF buffers (Section 3.5), leads to average `sync` times on the order of a few microseconds.

4 Correctness

We argue that nbMontage preserves the linearizability and lock freedom of a structure implemented on top of it, and adds buffered durable linearizability. We also argue that advances of the persistence frontier in nbMontage are wait free.

4.1 Linearizability

▶ **Theorem 1.** *Suppose that R is a data structure obeying the constraints of Section 3.2, running on nbMontage, and that K is realizable concrete history of R. K is linearizable.*

Proof (sketch). The `pnew`, `pdelete`, and `lin_CAS` routines of nbMontage have the same semantics as `new`, `delete`, and `CAS` calls in the original data structure. The `pdetach` routine has no semantic impact on crash-free execution: it simply ensures that a block whose removal has linearized will be reclaimed in post-crash recovery. The `sync` routine, similarly, has no semantic impact – with no parameters and no return values, it can linearize anywhere. If the instructions comprising each call to `pnew`, `pdelete`, `pdetach`, `sync`, and `lin_CAS` in a concrete nbMontage history are replaced with those of `new`, `delete`, `no-op`, `no-op`, and `CAS`, respectively, the result will be a realizable concrete history of the original data structure. Since that history is linearizable, so is the one on nbMontage. ◀

4.2 Buffered Durable Linearizability

As is conventional, we assume that each concurrent data structure implements some abstract data type. The semantics of such a type are defined in terms of legal *abstract sequential histories* – sequences of operations (request-response pairs), with their arguments and return values. We can define the *abstract state* of a data type, after some finite sequential history S, as the set of possible extensions to S permitted by the type's semantics. In a *concurrent* abstract history H, invocations and responses may be separated, and some responses may be missing, in which case the invocation is said to be *pending* at end of H. H is said to be *linearizable* if (1) there exists a history H' obtained by dropping some subset of the pending invocations in H and adding matching responses for the others, and (2) there exists a sequential history S that is equivalent to H' (same invocations and responses) and that respects both the *real-time* order of H' and the semantics of the abstract type. S is said to be a *linearization of H*.

Suppose now that R is a linearizable nonblocking implementation of type T, and that r is a concrete state of R – the bits in memory at the end of some concrete (instruction-by-instruction) history K. For R to be correct there must exist a mapping $\mathcal{M}$ such that for any such K and r, $\mathcal{M}(r)$ is the abstract state that results from performing, in order, the abstract operations corresponding to concrete operations that have linearized in K.

A structure R is *buffered durably linearizable* if post-crash recovery always results in some concrete state s that is justified by some prefix P of pre-crash concrete execution – that is, there exists a linearization S of the abstract history corresponding to P such that $\mathcal{M}(s)$ is the abstract state produced by S.

Consider again the 4 constraints listed at the end of Section 3.2 for data structures running on nbMontage. Elaborating on constraints 3 and 4, we use $r|_p$ to denote the set of payloads that were created (and inserted) and not yet detached by the operations that generated r. This allows us to recast constraint 4 and to add an additional constraint:

4'. There exists a mapping $\mathcal{Q}$ from sets of payloads to states of T such that for any concrete state r of R, $\mathcal{M}(r) = \mathcal{Q}(r|_p)$.

5. The recovery procedure of R, given a set of in-use payloads p, constructs a concrete state s such that $\mathcal{M}(s) = \mathcal{Q}(p)$.

▶ **Theorem 2.** *If a crash happens in epoch e, R will recover to a concrete state s such that $\mathcal{M}(s)$ is the abstract state produced by some linearization S of the abstract history H comprising pre-crash execution through the end of epoch $e-2$. In other words, R is buffered durably linearizable.*

Proof (sketch). For purposes of this proof, it is convenient to say that an update operation that commits the descriptor of its `lin_CAS` linearizes on the *preceding load* of the global epoch clock – the one that double-checks the clock before commit. Under this interpretation, if r is the concrete state of memory at the end of epoch $e-2$, we can say that $\mathcal{M}(r)$ reflects a sequential history containing all and only those operations that have committed their descriptors (line 17 in Fig. 8 in Full Version) by the end of the epoch. But this is not the only possible linearization of execution to that point! In particular, any operation that has loaded `global_epoch` (line 60 in Fig. 2), initialized its descriptor (line 65 of Fig. 2), and installed that descriptor in a `CASObj` (line 71 in Fig. 8 in Full Version) but has not yet committed the descriptor may "linearize in the past" (i.e., in epoch $e-2$) if it *or another, helping operation* commits the descriptor in the future. When a crash occurs in epoch e, any such retroactively linearized operations will see their payloads and anti-nodes included in the state s recovered from the crash. $\mathcal{M}(s)$ will therefore correspond, by constraint 5, to the linearization of execution through the end of epoch $e-2$ that includes all and only those pending operations that have linearized by the time of the crash. Crucially, if operation b depends on operation a, in the sense that a has completed in any extension of H in which b has completed, then, by constraint 2 of Section 3.2, the helping mechanism embodied by `lin_CAS` ensures that if b's payloads and anti-nodes are included in s, a's are included also. ◀

4.3 Wait-free Persistence

▶ **Theorem 3.** *The epoch advance in nbMontage is wait free.*

Proof (sketch). As shown in Fig. 2, an epoch advance from e to $e+1$ repeatedly finds a thread t that may still be active in $e-1$ (line 100), persists the contents of its TBP container and its descriptor (lines 101–102), and updates its mindicator entry. In the worst case, identifying all threads with blocks to be persisted requires time $O(T)$, where T is the number of threads, since the total size of the mindicator is roughly $2T$ nodes. Since each TBP container has bounded size, all the data of a thread can be persisted in $O(1)$ time. Mindicator updates, worst case, take $O(T \log T)$ time.

Since `sync` advances the epoch at most twice, it, too, is wait free. ◀

4.4 Lock freedom

▶ **Theorem 4.** *nbMontage preserves lock freedom during crash-free execution.*

Proof (sketch). Given Theorem 3, the only additional loop introduced by nbMontage is the automatic retry that occurs inside `lin_CAS` when the epoch has advanced. While this loop precludes wait freedom, we can (with a bit of effort – see App. C in Full Version) arrange to advance the epoch from e to $e+1$ only if some update operation has linearized in epoch $e-1$ or e (line 105 in Fig. 2). This suffices to preserve lock freedom. As a corollary, a data structure that is obstruction free remains so when persisted with nbMontage. ◀

5 Experimental Results

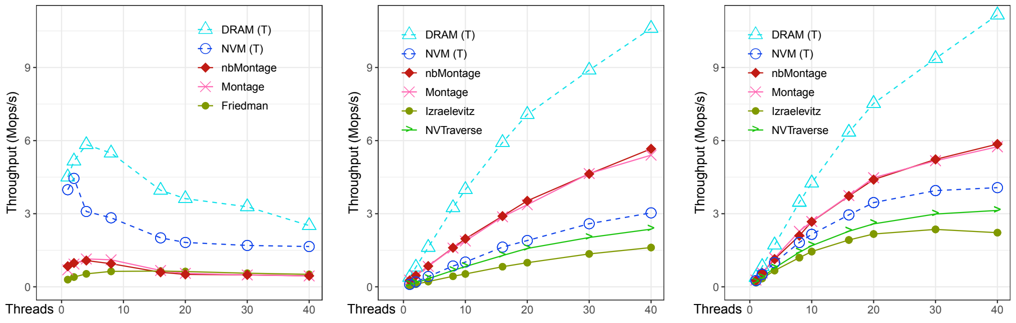
To confirm the performance benefits of buffering, we constructed nbMontage variants of Michael & Scott's queue [41], Natarajan & Mittal's binary search tree [42], the rotating skip list of Dick et al. [15], Michael's chained hash table [39], and the resizable hash table of Shalev & Shavit [48]. Mappings keep their key-value pairs in payloads and their index structures in transient memory. The queue uses payloads to hold values and their order. Specifically, between its two loads of the queue tail pointer, the enqueue operation calls `fetch_add` on a global counter to obtain a serial number for the to-be-inserted value. We benchmarked those data structures and various competitors on several different workloads. Below are the structures and systems we tested:

- **Montage** and **nbMontage** – as described in previous sections.
- **Friedman** – the persistent lock-free queue of Friedman et al. [19].
- **Izraelevitz** and **NVTraverse** – the N&M tree, the rotating skip list, and Michael's hash table persisted using Izraelevitz's transform [28] and the NVTraverse transform [18], respectively.
- **SOFT** and **NVMSOFT** – the lock-free hash table of Zuriel et al. [55], which persists only semantic data. SOFT keeps a full copy in DRAM, while NVMSOFT is modified to keep and access values only in NVM. Neither supports `update` on existing keys.
- **CLevel** – The persistent lock-free hash table of Chen et al. [8].
- **Dalí** – the lock-based buffered durably linearizable hash table of Nawab et al. [43].
- **DRAM (T)** and **NVM (T)** – as a baseline for comparison, these are unmodified transient versions of our various data structures, with data located in DRAM and NVM, respectively.

5.1 Configurations

We configured Montage and nbMontage with 64-entry TBP buffers and an epoch length of 10 ms. In practice, throughput is broadly insensitive to TBP size, and remains steady with epochs as short as 100 µs. All experiments were conducted on an Intel Xeon Gold 6230 processor with 20 physical cores and 40 hyperthreads, six 128 GB Optane Series 100 DIMMs, and six 32 GB DRAMs, running Fedora 30 Kernel 5.3.7 Linux Server. Threads are placed first on separate physical cores and then on hyperthreads. NVM is configured through the `dax-ext4` file system in "App Direct" mode.

All experiments use JEMalloc [16] for transient allocation and Ralloc [2] for persistent allocation, with the exception of CLevel, which requires the allocator from Intel's PMDK [49]. All chained hash tables have 1 million buckets. The warm-up phase for mappings inserts 0.5 M key-value pairs drawn from a key space of 1 M keys. Queues are initialized with 2000 items. Unless otherwise specified, keys and values are strings of 32 and 1024 bytes, respectively. We report the average of three trials, each of which runs for 30 seconds. Source code for nbMontage and the experiments is available at `https://github.com/urcs-sync/Montage`.

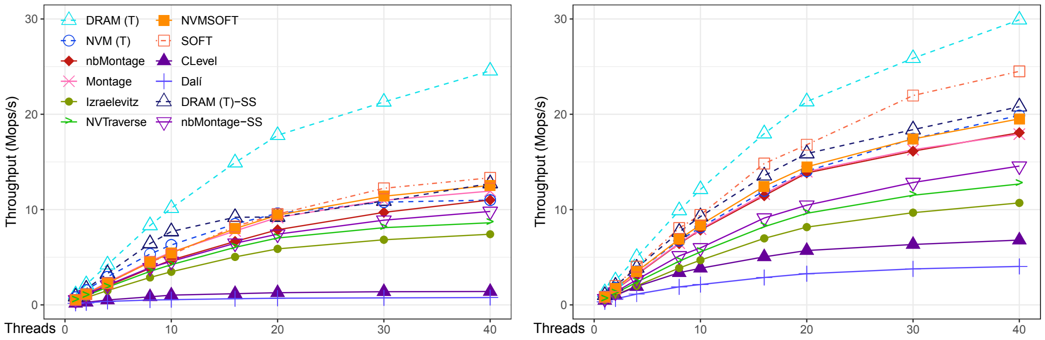


(a) Queues – 1:1 push:pop.

(b) Binary search trees – 2:1:1 get:insert:remove.

(c) Skip lists – 2:1:1 get:insert: remove.

Figure 3 Throughput of concurrent queues, binary search trees, and skip lists.



(a) 2:1:1 get:insert:remove.

(b) 18:1:1 get:insert:remove.

Figure 4 Hash table throughput. Options in the left column of the key are all variants of Michael's nonblocking algorithm.

5.2 Throughput

Results for queues, binary search trees, skip lists, and hash tables appear in Figures 3–4. The nbMontage versions of the M&S queue, N&M tree, rotating skip list, and Michael hash table outperform most persistent alternatives by a significant margin – up to 2× faster than Friedman et al.'s queue, 1.3–4× faster than NVTraverse and Izraelevitz et al.'s transform, and 3–14× faster than CLevel and Dalí. Significantly, nbMontage achieves almost the same throughput as Montage. SOFT and NVMSOFT are the only exceptions: the former benefits from keeping a copy of its data in DRAM; both benefit from clever internal optimizations. The DRAM copy has the drawback of forgoing the extra capacity of NVM; the optimization has the drawback of precluding atomic `update` of existing keys. While the transient Shalev & Shavit (S&S) hash table (DRAM (T)-SS in Fig. 4) is significantly slower than the transient version of Michael's hash table (DRAM (T)), the throughput of the Montage version (nbMontage-SS) is within 65% of the transient version and still faster than all other pre-existing persistent options other than SOFT and NVMSOFT.

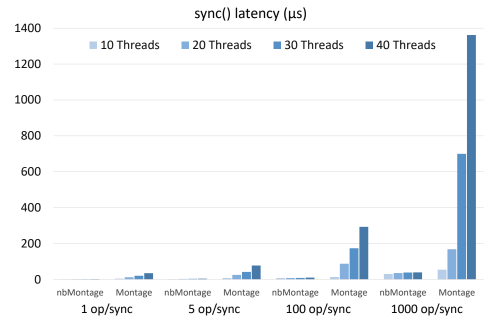


Figure 5 Average latency of `sync` on hash tables.

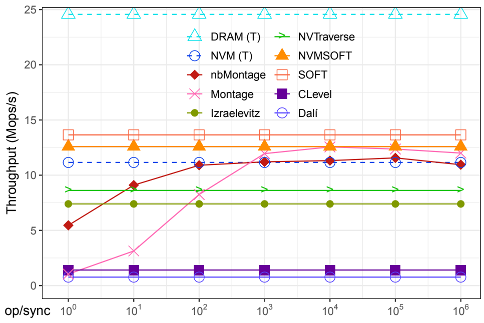


Figure 6 Throughput of hash tables with a `sync` every x operations on average.

5.3 Overhead of Sync

To assess the efficacy of nonblocking epoch advance and of our mindicator variant, we measured the latency of `sync` and the throughput of code that calls `sync` frequently. Specifically, using the nbMontage version of Michael's hash table, on the (2:1:1 get:insert:remove) workload, we disabled the periodic epoch advance performed by a background thread and had each worker call `sync` explicitly.

Average `sync` latency is shown in Figure 5 for various thread counts and frequencies of calls, on both nbMontage and its blocking predecessor. In all cases, nbMontage completes the typical `sync` in 1–40 µs. In the original Montage, however, `sync` latency never drops below 5 µs, and can be as high as 1.3 ms with high thread count and low frequency.

Hash table throughput as a function of `sync` frequency is shown in Figure 6 for 40 threads running on Montage and nbMontage. For comparison, horizontal lines are shown for various persistent alternatives (none of which calls `sync`). Interestingly, nbMontage is more than 2× faster than CLevel even when `sync` is called in every operation, and starts to outperform NVTraverse once there are more than about 10 operations per `sync`.

5.4 Recovery

To assess recovery time, we initialized the nbMontage version of Michael's hash table with 1–32 M 1 KB elements, leading to a total payload footprint of 1–32 GB. With one recovery thread, nbMontage recovered the 1 GB data set in 1.4 s and the 32 GB data set in 103.8 s (22.2 s to retrieve all in-use blocks; 81.6 s to insert them into a fresh hash table). Eight recovery threads accomplished the same tasks in 0.3 s and 17.9 s. These times are all within 0.5 s of recovery times on the original Montage.

6 Conclusions

To the best of our knowledge, nbMontage is the first general-purpose system to combine buffered durable linearizability with a simple API for nonblocking data structures and nonblocking progress of the persistence frontier. Nonblocking persistence allows nbMontage to provide a very fast wait-free `sync` routine and to strictly bound the work that may be lost on a crash. Lock-free and wait-free structures, when implemented on nbMontage, remain lock free; obstruction-free structures remain obstruction free.

Experience with a variety of nonblocking data structures confirms that they are easy to port to nbMontage, and perform extremely well – better in most cases than even hand-built structures that are strictly durably linearizable. Given that programmers have long been accustomed to `sync`-ing their updates to file systems and databases, a system with the performance and formal guarantees of nbMontage appears to be of significant practical utility. In ongoing work, we are exploring the design of a hybrid system that supports both lock-based and nonblocking structures, with nonblocking persistence in the absence of lock-based operations. We also hope to develop a buffered durably linearizable system for object-based software transactional memory, allowing persistent operations on multiple data structures to be combined into failure-atomic transactions.

References

1 H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. Nonblocking persistent software transactional memory. In *27th Intl. Conf. on High Performance Computing, Data, and Analytics (HiPC)*, pages 283–293, 2020. doi:10.1109/HiPC50609.2020.00042.

2 Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *19th Intl. Symp. on Memory Management (ISMM)*, 2020. doi:10.1145/3381898.3397212.

3 Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. B3-tree: Byte-addressable binary B-tree for persistent memory. *ACM Trans. on Storage*, 16(3):17:1–17:27, July 2020. doi:10.1145/3394025.

4 Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 433–452, Portland, OR, October 2014. doi:10.1145/2660193.2660224.

5 Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. of the VLDB Endowment*, 8(5):497–508, 2015. doi:10.14778/2735479.2735483.

6 Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. of the VLDB Endowment*, 8(7):786–797, February 2015. doi:10.14778/2752939.2752947.

7 Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A persistent B+-tree with low tail latency. *Proc. of the VLDB Endowment*, 13(11):2634–2648, August 2020. doi:10.14778/3407790.3407850.

8 Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *Usenix Annual Technical Conf. (ATC)*, pages 799–812, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/chen.

9 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, Newport Beach, CA, 2011. doi:10.1145/1950365.1950380.

10 Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *24th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 441–454, Providence, RI, USA, 2019. doi:10.1145/3297858.3304046.

11 Nachshon Cohen, David T. Aksun, and James R. Larus. Object-oriented recovery for non-volatile memory. *Proc. of the ACM on Programming Languages*, 2(OOPSLA):153:1–153:22, October 2018. doi:10.1145/3276523.

12 Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *30th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, page 259–269, Vienna, Austria, 2018. doi:10.1145/3210377.3210400.

13 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *30th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 271–282, Vienna, Austria, July 2018. doi:10.1145/3210377.3210392.

14 Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Usenix Annual Technical Conf. (ATC)*, pages 373–386, Boston, MA, 2018. URL: `https://www.usenix.org/conference/atc18/presentation/david`.

15 Ian Dick, Alan Fekete, and Vincent Gramoli. A skip list for multicore. *Concurrency and Computation: Practice and Experience*, 29(4), May 2016.

16 Jason Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conf.*, Ottawa, ON, Canada, May 2006. URL: `https://papers.freebsd.org/2006/bsdcan/evans-jemalloc.files/evans-jemalloc-paper.pdf`.

17 Keir Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, Univ. of Cambridge, 2003. Published as Univ. of Cambridge Computer Laboratory technical report #579, February 2004. `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf`.

18 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *41st ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 377–392, 2020. `doi:10.1145/3385412.3386031`.

19 Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, Vienna, Austria, 2018. `doi:10.1145/3178487.3178490`.

20 Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, HTM-compatible persistent transactions. In *41st ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 59–74, June 2020. `doi:10.1145/3385412.3385991`.

21 Ellis R. Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *31st Symp. on Mass Storage Systems and Technologies (MSST)*, pages 1–14, Santa Clara, CA, May–June 2015. `doi:10.1109/MSST.2015.7208276`.

22 Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Usenix Annual Technical Conf. (ATC)*, pages 913–928, Renton, WA, July 2019. URL: `https://www.usenix.org/conference/atc19/presentation/gu`.

23 Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *16th Intl. Symp. on Distributed Computing (DISC)*, pages 265–279, Toulouse, France, October 2002. `doi:10.1007/3-540-36108-1_18`.

24 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990. `doi:10.1145/78969.78972`.

25 Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *12th European Conf. on Computer Systems (EuroSys)*, pages 468–482, Belgrade, Serbia, 2017. `doi:10.1145/3064176.3064204`.

26 Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *16th Usenix Conf. on File and Storage Technologies (FAST)*, pages 187–200, Oakland, CA, February 2018. URL: `https://www.usenix.org/conference/fast18/presentation/hwang`.

27 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, Atlanta, GA, 2016. `doi:10.1145/2872362.2872410`.

28 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Intl. Symp. on Distributed Computing (DISC)*, pages 313–327, Paris, France, September 2016. `doi:10.1007/978-3-662-53426-7_23`.

29 Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Trans. on Storage*, 14(1):5:1–5:17, February 2018. doi:10.1145/3129263.

30 Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th Usenix Conf. on File and Storage Technologies (FAST)*, pages 257–270, Santa clara, CA, February 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon.

31 Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. Building scalable NVM-based B+tree with HTM. In *48th Intl. Conf. on Parallel Processing (ICPP)*, pages 101:1–101:10, Kyoto, Japan, August 2019. doi:10.1145/3337821.3337827.

32 Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–343, Xi'an, China, April 2017. doi:10.1145/3037697.3037714.

33 Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Intl. Symp. on Microarchitecture (MICRO)*, pages 258–270, Fukuoka, Japan, 2018. doi:10.1109/MICRO.2018.00029.

34 Yujie Liu, Victor Luchangco, and Michael Spear. Mindicators: A scalable approach to quiescence. In *IEEE 33rd Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 206–215, Philadelphia, PA, 2013. doi:10.1109/ICDCS.2013.39.

35 Pratyush Mahapatra, Mark D Hill, and Michael M Swift. Don't persist all: Efficient persistent data structures, 2019. arXiv preprint. arXiv:1905.13011.

36 Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *7th European Conf. on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, April 2012. doi:10.1145/2168836.2168855.

37 Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *25th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 789–806, March 2020. doi:10.1145/3373376.3378456.

38 Amirsaman Memaripour and Steven Swanson. Breeze: User-level access to non-volatile main memories for legacy software. In *36th Intl. Conf. on Computer Design (ICCD)*, pages 413–422, Hartford, CT, October 2018. doi:10.1109/ICCD.2018.00069.

39 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *14th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82, Winnipeg, MB, Canada, 2002. doi:10.1145/564870.564881.

40 Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi:10.1109/TPDS.2004.8.

41 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, 1996. doi:10.1145/248052.248106.

42 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, Orlando, FL, USA, February 2014. doi:10.1145/2555243.2555256.

43 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *Intl. Symp. on Distributed Computing (DISC)*, pages 37:1–37:16, Vienna, Austria, 2017. doi:10.4230/LIPIcs.DISC.2017.37.

44 Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Intl Conf on Management of Data (SIGMOD)*, pages 371–386, San Francisco, CA, 2016. `doi:10.1145/2882903.2915251`.

45 Matej Pavlovic, Alex Kogan, Virendra J Marathe, and Tim Harris. Brief announcement: Persistent multi-word compare-and-swap. In *ACM Symp. on Principles of Distributed Computing (PODC)*, pages 37–39, Egham, United Kingdom, 2018. `doi:10.1145/3212734.3212783`.

46 Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. OneFile: A wait-free persistent transactional memory. In *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 151–163, Portland, OR, June 2019. `doi:10.1109/DSN.2019.00028`.

47 David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *3rd VLDB Workshop on In-Memory Data Management and Analytics (IMDM)*, pages 4:1–4:8, Kohala, HI, 2015. `doi:10.1145/2803140.2803144`.

48 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, pages 379–405, May 2006. `doi:10.1145/1147954.1147958`.

49 Usharani Upadhyayula and Andy M. Rudoff. Introduction to Programming with Persistent Memory from Intel. `https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel`, August 2017.

50 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th Usenix Conf. on File and Storage Technologies (FAST)*, pages 61–75, San Jose, CA, 2011. URL: `http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman`.

51 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, Newport Beach, CA, 2011. `doi:10.1145/1950365.1950379`.

52 Chundong Wang, Qingsong Wei, Lingkun Wu, Sibo Wang, Cheng Chen, Xiaokui Xiao, Jun Yang, Mingdi Xue, and Yechao Yang. Persisting RB-Tree into NVM in a consistency perspective. *ACM Trans. on Storage*, 14(1):6:1–6:27, February 2018. `doi:10.1145/3177915`.

53 Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. A fast, general system for buffered persistent data structures. In *50th Intl. Conf. on Parallel Processing (ICPP)*, August 2021. URL: `https://oaciss.uoregon.edu/icpp21/views/includes/files/pap118s4-file2.pdf`.

54 Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th Usenix Conf. on File and Storage Technologies (FAST)*, pages 167–181, Santa Clara, CA, February 2015. URL: `https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang`.

55 Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proc. of the ACM on Programming Languages*, 3(OOPSLA):128:1–128:26, October 2019. `doi:10.1145/3360554`.

A Example nbMontage Data Structure

As an example of using the nbMontage API, Fig. 7 presents a fragment of Michael's lock-free hash table [39], modified for persistence. Highlighted parts were changed from the original.

```
class MHashTable :  public Recoverable {
class Payload :  public PBlk { K key; V val; };
struct Node { // Transient index class
  Payload* payload = nullptr; // Transient-to-persistent pointer
  CASObj<Node*> next = nullptr; // Transient-to-transient pointer
  Node(K& key, V& val) { payload = pnew<Payload>(key, val); }
  ~Node() { if(payload!=nullptr) pdelete(payload); }
};
EBRTracker tracker; // Epoch-based memory reclamation
bool find(CASObj<Node*>* &p,Node* &c,Node* &n,K k); // Starting from p, find node >= k and assign to c
void put(K key, V val) { // Insert, or update if the key exists
  Node* new_node = new Node(key, val);
  CASObj<Node*>* prev = nullptr;
  Node* curr;
  Node* next;
  tracker.start_op();
  while(true) {
    if (find(prev,curr,next,key)) { // update
      new_node->next.store(curr);
      pdetach(curr->payload);
      if(prev->lin_CAS(curr,new_node)) {
        while(!curr->next.CAS(next,mark(next))) next=curr->next.load();
        if(new_node->next.CAS(curr,next)) tracker.retire(curr);
        else find(prev,curr,next,key);
        break;
      }
    } else { // key does not exist; insert
      new_node->next.store(curr);
      if(prev->lin_CAS(curr,new_node))
        break;
    }
  }
  tracker.end_op();
};
```

Figure 7 Michael's lock-free hash table example (nbMontage-related parts highlighted).

Massively Parallel Correlation Clustering in Bounded Arboricity Graphs

Mélanie Cambus
Aalto University, Finland

Davin Choo
National University of Singapore, Singapore

Havu Miikonen
Aalto University, Finland

Jara Uitto
Aalto University, Finland

Abstract

Identifying clusters of similar elements in a set is a common task in data analysis. With the immense growth of data and physical limitations on single processor speed, it is necessary to find efficient parallel algorithms for clustering tasks. In this paper, we study the problem of correlation clustering in bounded arboricity graphs with respect to the Massively Parallel Computation (MPC) model. More specifically, we are given a complete graph where the edges are either positive or negative, indicating whether pairs of vertices are similar or dissimilar. The task is to partition the vertices into clusters with as few disagreements as possible. That is, we want to minimize the number of positive inter-cluster edges and negative intra-cluster edges.

Consider an input graph G on n vertices such that the positive edges induce a λ-arboric graph. Our main result is a 3-approximation (*in expectation*) algorithm to correlation clustering that runs in $\mathcal{O}(\log \lambda \cdot \operatorname{poly}(\log \log n))$ MPC rounds in the *strongly sublinear memory regime*. This is obtained by combining structural properties of correlation clustering on bounded arboricity graphs with the insights of Fischer and Noever (SODA '18) on randomized greedy MIS and the `PIVOT` algorithm of Ailon, Charikar, and Newman (STOC '05). Combined with known graph matching algorithms, our structural property also implies an exact algorithm and algorithms with *worst case* $(1+\varepsilon)$-approximation guarantees in the special case of forests, where $\lambda = 1$.

2012 ACM Subject Classification Theory of computation → MapReduce algorithms; Theory of computation → Unsupervised learning and clustering

Keywords and phrases MPC Algorithm, Correlation Clustering, Bounded Arboricity

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.15

Related Version *Full Version*: `https://arxiv.org/abs/2102.11660`

Funding *Mélanie Cambus*: This work was supported in part by the Academy of Finland, Grant 334238.
Davin Choo: This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG-PhD/2021-08-013).
Jara Uitto: This work was supported in part by the Academy of Finland, Grant 334238.

1 Introduction

Graphs are a versatile abstraction of datasets and clustering on graphs is a common unsupervised machine learning task for data-analytical purposes such as community detection and link prediction [14, 11].

Here, we study the correlation clustering problem which aims at grouping elements of a dataset according to their similarities. Consider the setting where we are given a complete signed graph $G = (V, E = E^+ \cup E^-)$ where edges are given positive (E^+) or negative (E^-)

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 15; pp. 15:1–15:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

labels, signifying whether two points are similar or not. The task is to find a partitioning of the vertex set V into clusters $C_1, C_2, \ldots, C_r$, where r is not fixed by the problem statement but can be chosen freely by the algorithm.[1] If endpoints of a positive edge belong to the same cluster, we say that the edge is a *positive agreement*; and a *positive disagreement* otherwise. Meanwhile, if endpoints of a negative edge belong to the same cluster, we say that the edge is a *negative disagreement*; and a *negative agreement* otherwise. The goal of correlation clustering is to obtain a clustering that maximizes agreements or minimizes disagreements.

As pointed out by Chierichetti, Dalvi and Kumar [15], the positive degrees of vertices are typically bounded in many applications. This motivates the study of parallel algorithms for correlation clustering as a function of the maximum degree of the input graph. However, many real life networks, such as those modelled by scale-free network models (such as Barabási-Albert), admit structures with a few high degree nodes and a small average degree. To capture such graphs, we generalize the study of bounded degree graphs to the study of low arboricity graphs in this work. In particular, we focus on the case of minimizing disagreements when the positive edges of the input graph induces a λ-arboric graph.

In the complete signed graph setting, one can perform cost-charging arguments via "bad triangles" to prove approximation guarantees. A set of 3 vertices $\{u, v, w\}$ is a *bad triangle* if $\{u, v\}, \{v, w\} \in E^+$ and $\{u, w\} \in E^-$. As edges of any bad triangle induce at least one disagreement in any clustering, one can lower bound the cost of any optimum clustering by the number of *edge-disjoint* bad triangles in the input graph. `PIVOT` [2] is a well-known algorithm that provides a 3-approximation (in expectation) to the problem of minimizing disagreements in the sequential setting by using a cost-charging argument on bad triangles. It works as follows: as long as the graph is non-empty, pick a vertex v uniformly at random and form a new cluster using v and its "positive neighbors" (i.e. joined by a positive edge). One can view `PIVOT` as simulating greedy MIS with respect to a uniform-at-random permutation of vertices.[2]

Many of the known distributed algorithms for the correlation clustering problem adapt the `PIVOT` algorithm. The basic building block is to fix a random permutation and to create the clusters by finding, in parallel, local minimums according to the permutation. The `ParallelPIVOT`, `C4` and `ClusterWild!` algorithms [15, 33] all obtain constant approximations in $\mathcal{O}(\log n \cdot \log \Delta)$ synchronous rounds, where Δ stands for the maximum positive degree.[3] Meanwhile, with a tighter analysis of randomized greedy MIS algorithm [21], one can obtain a 3-approximation in $\mathcal{O}(\log n)$ rounds by directly simulating `PIVOT`. All above approximation guarantees are in expectation.

1.1 Computational model

We consider the *Massive Parallel Computation (MPC) model* [30, 8] which serves as a theoretical abstraction of several popular massively parallel computation frameworks such as Dryad [29], Hadoop [36], MapReduce [18], and Spark [37].

[1] This is in contrast to, for example, the classic k-means clustering where k is an input problem parameter.

[2] A subset $M \subseteq V$ is a maximal independent set (MIS) if (1) for any two vertices $u, v \in M$, u and v are not neighbors, and (2) for any vertex $v \in V$, either $v \in M$ or v has a neighbor in M. Given a vertex ordering $\pi : [n] \to V$, greedy MIS refers to the process of iterating through $\pi(1), \ldots, \pi(n)$ and adding each vertex to M if it has no neighbor of smaller ordering.

[3] Technically speaking, `ParallelPIVOT` does not compute a greedy MIS. Instead, it computes random independent sets in each phase and only uses the initial random ordering to perform tie-breaking. i.e. if a vertex u has more than one positive neighbor in the independent set, then vertex u joins the cluster defined by the neighbor with the smallest assigned order.

In the MPC model, we have M machines, each with memory size S, and we wish to solve a problem given an input of size N. In the context of correlation clustering, we may think of $N = |E^+|$, since the negative edges can be inferred from missing positive edges. Typically, the *local* memory bound S is assumed to be significantly smaller than N. We focus on the *strongly sublinear memory* regime, where $S = \widetilde{\mathcal{O}}\left(n^\delta\right)$ for some constant $\delta < 1$. Ideally, the total memory $S \cdot M$ is not much larger than N.

The computation in the MPC model proceeds in *synchronous rounds*. In each round, each machine can perform arbitrary computation on the data that resides on it.[4] Then, each machine communicates in an all-to-all fashion with all other machines conditioned on sending and receiving messages of size at most $\mathcal{O}(S)$. This concludes the description of an MPC round. Since communication costs are typically the bottleneck, the metric for evaluating the efficiency of an MPC algorithm is the *number of rounds* required.

1.2 Our contributions

Our goal is to obtain efficient algorithms for correlation clustering in the sublinear memory regime of MPC (see Model 1) when given a complete signed graph G, with maximum positive degree Δ, where the set of positive edges E^+ induces a λ-arboric graph. Our main contributions are the following:

1. By combining known techniques, we show that one can compute a randomized greedy MIS, with respect to a uniform-at-random permutation of vertices, in $\mathcal{O}\left(\log \Delta \cdot \log^3 \log n\right)$ MPC rounds. If we allow extra global memory (see Model 2), this can be sped up to $\mathcal{O}\left(\log \Delta \cdot \log \log n\right)$ MPC rounds. See Theorem 6 for details.
   We believe that this result is of independent interest beyond applications to correlation clustering. To the best of our knowledge, our algorithm for greedy MIS improves upon the state-of-the-art for any $\Delta \in o(n^{1/\log^3 \log n})$.
2. Our main result (Theorem 12) is that one can effectively ignore vertices of degrees larger than $\mathcal{O}(\lambda)$ when computing a correlation clustering. Then, the overall runtime and approximation guarantees are inherited from the choice of algorithm used to solve correlation clustering on the remaining bounded degree subgraph.[5]
3. Using our main result, we show how to obtain efficient correlation clustering algorithms for bounded arboricity graphs. By simulating `PIVOT` on a graph with maximum degree $\mathcal{O}(\lambda)$ via Theorem 6, we get
   (i) A 3-approximation (in expectation) algorithm in $\mathcal{O}\left(\log \lambda \cdot \log^3 \log n\right)$ MPC rounds.
   (ii) A 3-approximation (in expectation) algorithm in $\mathcal{O}\left(\log \lambda \cdot \log \log n\right)$ MPC rounds, possibly using extra global memory.
   In the special case of forests (where $\lambda = 1$), we show that the optimum correlation clustering is equivalent to computing a *maximum matching*. Let $0 < \varepsilon \leq 1$ be a constant. By invoking three different known algorithms (one for *maximum matching* and two for *maximal matching*), and hiding $1/\varepsilon$ factors in $\mathcal{O}_\varepsilon(\cdot)$, we obtain
   (iii) An exact randomized algorithm that runs in $\widetilde{\mathcal{O}}\left(\log n\right)$ MPC rounds.
   (iv) A $(1+\varepsilon)$-approx. (worst case) det. algo. that runs in $\mathcal{O}_\varepsilon\left(\log \log^* n\right)$ MPC rounds.
   (v) A $(1+\varepsilon)$-approx. (worst case) randomized algo. that runs in $\mathcal{O}_\varepsilon(1)$ MPC rounds.
   Finally, for low-arboricity graphs, the following result may be of interest:
   (vi) An $\mathcal{O}\left(\lambda^2\right)$-approx. (worst case) deterministic algo. that runs in $\mathcal{O}(1)$ MPC rounds.

For more details and an in-depth discussion about our techniques, see Section 2.

[4] Although there is no hard computation constraint in the MPC model, all known MPC algorithms spend polynomial time on each machine in any given round.

[5] In some works, "bounded degree" is synonymous with "maximum degree $\mathcal{O}(1)$". Here, we mean that the maximum degree is $\mathcal{O}(\lambda)$.

1.3 Outline and notation

1.3.1 Outline

Before diving into formal details, we highlight the key ideas behind our results in Section 2. Section 3 shows how to efficiently compute a randomized greedy MIS. Our structural result about correlation clustering in bounded arboricity graphs is presented in Section 4. We combine this structural insight with known algorithms in Section 5 to yield efficient correlation clustering algorithms. Finally, we conclude with some open questions in Section 6.

Please see the full version[6] for formal proofs.

1.3.2 Notation

In this work, we only deal with complete signed graphs $G = (V, E = E^+ \cup E^-)$ where $|V| = n$, $|E| = \binom{n}{2}$, and E^+ and E^- denote the sets of positively and negatively labeled edges respectively. For a vertex v, the sets $N^+(v) \subseteq V$ and $N^-(v) \subseteq V$ denote vertices that are connected to v via positive and negative edges, respectively. We write $\Delta = \max_{v \in V} |N^+(v)|$ as the maximum *positive* degree in the graph. The k-hop neighborhood of a vertex v is the set of vertices that have a path from v involving at most k *positive* edges.

A clustering $\mathcal{C}$ is a partition of the vertex set V. That is, $\mathcal{C}$ is a set of sets of vertices such that (i) $A \cap B = \emptyset$ for any two sets $A, B \in \mathcal{C}$ and (ii) $\cup_{A \in \mathcal{C}} A = V$. For a cluster $C \subseteq V$, $N_C^+(v) = N^+(v) \cap C$ is the set of positive neighbors of v that lie within cluster C. We write $d_C^+(v) = |N^+(v) \cap C|$ to denote the positive degree of v within C. If endpoints of a positive edge *do not* belong to the same cluster, we say that the edge is a *positive disagreement*. Meanwhile, if endpoints of a negative edge belong to the same cluster, we say that the edge is a *negative disagreement*. Given a clustering $\mathcal{C}$, the cost of a clustering $cost(\mathcal{C})$ is defined as the total number of disagreements.

The arboricity λ_G of a graph $G = (V, E)$ is defined as $\lambda_G = \max_{S \subseteq V} \left\lceil \frac{|E(S)|}{|S|-1} \right\rceil$, where $E(S)$ is the set of edges induced by $S \subseteq V$. We drop the subscript G when it is clear from context. A graph with arboricity λ is said to be λ-arboric. We denote the set $\{1, 2, \dots, n\}$ by $[n]$. We hide absolute constant multiplicative factors and multiplicative factors logarithmic in n using standard notations: $\mathcal{O}(\cdot)$, $\Omega(\cdot)$, and $\widetilde{\mathcal{O}}(\cdot)$. The notation $\log^* n$ refers to the smallest integer t such that the t-iterated logarithm of n is at most 1.[7] An event $\mathcal{E}$ on a n-vertex graph holds with high probability if it happens with probability at least $1 - n^{-c}$ for an arbitrary constant $c > 1$, where c may affect other constants (e.g. those hidden in the asymptotics).

We now fix the parameters in our model of computation. Model 1 is the standard definition of strongly sublinear MPC regime while Model 2 is a relaxed variant which guarantees that there are at least $M \geq n$ machines. While the latter model may utilize more global memory than the standard strongly sublinear regime, it facilitates conceptually simpler algorithms.

▶ **Model 1** (Strongly sublinear MPC regime). *Consider the MPC model. The input graph with n vertices is of size $N \in \Omega(n)$. We have $M \in \Omega(N/S)$ machines, each having memory size $S \in \widetilde{\mathcal{O}}(n^\delta)$, for some constant $0 < \delta < 1$. The total global memory usage is $M \cdot S \geq N$.*

▶ **Model 2** (Strongly sublinear MPC regime with at least n machines). *Consider the MPC model. The input graph with n vertices is of size $N \in \Omega(n)$. We have $M \geq n$ machines and each vertex is given access to a machine with memory size $S \in \widetilde{\mathcal{O}}(n^\delta)$, for some constant $0 < \delta < 1$. The total global memory usage is $\max\{\widetilde{\mathcal{O}}(n^{1+\delta}), M \cdot S\} \geq N$.*

[6] Available at https://arxiv.org/abs/2102.11660.

[7] That is, $\log^{(t)} n \leq 1$. For all practical values of n, one may treat $\log^* n \leq 5$.

To avoid unnecessary technical complications for Model 2, we assume throughout the paper that $\Delta \in \mathcal{O}(S)$. This assumption can be lifted using the virtual communication tree technique described by Ghaffari and Uitto [26].

▶ Remark 3 (Role and motivation for Model 2). From an algorithmic design perspective, the slightly relaxed Model 2 allows one to focus on keeping the amount of "local memory required by each vertex" to the sublinear memory regime. Oftentimes[8], algorithms are first described in relaxed models (such as Model 2, or by simply allowing more total global memory used) with a simple-to-understand analysis before using further complicated argument/analysis to show that it in fact also works in Model 1.[9]

1.4 Further related work

Correlation clustering on complete signed graphs was introduced by Bansal, Blum and Chawla [4].[10] They showed that computing the optimal solution to correlation clustering is NP-complete, and explored two different optimization problems: maximizing agreements, or minimizing disagreements. While the optimum clusterings to both problems are the same (i.e. a clustering minimizes disagreements if and only if it maximizes agreements), the complexity landscapes of their approximate versions are wildly different.

Maximizing agreements is known to admit a polynomial time approximation scheme in complete graphs [4]. Furthermore, Swamy [35] gave a 0.7666-approximation on general weighted graphs via semidefinite programming.

On the other hand, for minimizing disagreements, the best known approximation ratio for complete graphs is 2.06, due to CMSY [13], via probabilistic rounding of a linear program (LP) solution. This 2.06-approximation uses the same LP as the one proposed by Ailon, Charikar and Newman [2] but performs probabilistic rounding more carefully, nearly matching the integrality gap of 2 shown by Charikar, Guruswami and Wirth [12]. In general weighted graphs, the current state of the art, due to DEFI [19], gives an $\mathcal{O}(\log n)$-approximation through an LP rounding scheme.

In a distributed setting, PPORRJ [33] presented two random algorithms (`C4` and `ClusterWild!`) to address the correlation clustering problem in the case of complete graphs, aiming at better time complexities than `KwikCluster`. The `C4` algorithm gives a 3-approximation in expectation, with a polylogarithmic number of rounds where the greedy MIS problem is solved on each round. The `ClusterWild!` algorithm gives up on the independence property in order to speed up the process, resulting in a $(3 + \varepsilon)$-approximation. Both those algorithms are proven to terminate after $\mathcal{O}\left(\frac{1}{\epsilon} \cdot \log n \cdot \log \Delta\right)$ rounds with high probability. A third distributed algorithm for solving correlation clustering is given by Chierichetti, Dalvi and Kumar [15] for the MapReduce model. Their algorithm, `ParallelPivot`, also gives

[8] As a warm-up description of their algorithm, Ghaffari and Uitto [26, Assumption (2) on page 6] uses more machines than just $M = N/S$. Meanwhile, using slightly more global memory, the algorithm of ASSWZ [3] is straightforward to understand (e.g. see Ghaffari [22, Section 3.3]) and can achieve conjecturally optimal running time, with respect to the $\Omega(\log D)$ conditional lower bound for solving graph connectivity in MPC via the 2-cycle problem.

[9] In our case, we first designed Algorithm 3 in Model 2 but were unable to show that it also works in Model 1. Thus, we designed Algorithm 2 that works in Model 1. However, we decided to keep the description and analysis of the simpler Algorithm 3 in the paper – it is algorithmically very clean and we hope that it is easier to understand the technicalities of the more involved Algorithm 3 after seeing the structure and analysis of the simpler Algorithm 2.

[10] For relevant prior work, we try our best to list all authors when there are three or less, and use their initials when there are more (e.g. CMSY, PPORRJ, BBDFHKU). While this avoids the use of et al. in citations in favor of an equal mention of all authors' surnames, we apologize for the slight unreadability.

a constant approximation in polylogarithmic time, without solving a greedy MIS in each round. Using a tighter analysis, Fischer and Noever [21] showed that randomized greedy MIS terminates in $\mathcal{O}(\log n)$ round with high probability, which directly implies an $\mathcal{O}(\log n)$ round simulation of `PIVOT` in various distributed computation models.

For our approach, the *randomized greedy MIS* plays a crucial role in terms of the approximation ratio. Blelloch, Fineman and Shun [10] showed that randomized greedy MIS terminates in $\mathcal{O}(\log^2 n)$ parallel rounds with high probability. This was later improved to $\mathcal{O}(\log n)$ rounds by Fischer and Noever [21]. Faster algorithms are known for finding an MIS that may not satisfy the greedy property. For example, Ghaffari and Uitto [26] showed that there is an MIS algorithm running in $\mathcal{O}\left(\sqrt{\log \Delta} \cdot \log\log \Delta + \sqrt{\log\log n}\right)$ MPC rounds. This algorithm was later adapted to bounded arboricity with runtime of $\mathcal{O}\left(\sqrt{\log \lambda} \cdot \log\log \lambda + \log^2 \log n\right)$ by BBDFHKU [9] and improved to $\mathcal{O}\left(\sqrt{\log \lambda} \cdot \log\log \lambda + \log\log n\right)$ by Ghaffari, Grunau and Jin [24]. There is also a deterministic MIS algorithm that runs in $\mathcal{O}\left(\log \Delta + \log\log n\right)$ MPC rounds due to Czumaj, Davies and Parter [16].

2 Techniques

In this section, we highlight the key ideas needed to obtain our results described in Section 1.2. We begin by explaining some computational features of the MPC model so as to set up the context needed to appreciate our algorithmic results. By exploiting these computational features together with a structural result of randomized greedy MIS by Fischer and Noever [21], we explain how to compute a randomized greedy MIS in $\mathcal{O}\left(\log \Delta \cdot \log\log n\right)$ MPC rounds. We conclude this section by explaining how to obtain our correlation clustering results by using our structural lemma that reduces the maximum degree of the input graph to $\mathcal{O}(\lambda)$.

2.1 Computational features of MPC

2.1.1 Detour: The classical models of LOCAL and CONGEST

To better appreciate of the computational features of MPC, we first describe the classical distributed computational models of LOCAL and CONGEST [32, 34].

In the LOCAL model, all vertices are treated as individual computation nodes and are given a unique identifier – some binary string of length $\mathcal{O}(\log n)$. Computation occurs in synchronous rounds where each vertex does the following: perform arbitrary local computations, then send messages (of unbounded size) to neighbors. As the LOCAL model does not impose any restrictions on computation or communication costs (beyond a topological restriction), the performance of LOCAL algorithms is measured in the number of rounds used. Furthermore, since nodes can send unbounded messages, every vertex can learn about its k-hop neighborhood in k LOCAL rounds.

The CONGEST model is identical to the LOCAL model with an additional restriction: the size of messages that can be sent or received per round can only be $\mathcal{O}(\log n)$ bits across each edge. This means that CONGEST algorithms may no longer assume that they can learn about the k-hop topology for every vertex in k CONGEST rounds.

Since the MPC model does not restrict computation within a machine, one can directly simulate any k-round LOCAL or CONGEST algorithm in $\mathcal{O}(k)$ MPC rounds, as long as each machine sends and receives messages of size at most $\mathcal{O}(S)$. This often allows us to directly invoke existing LOCAL and CONGEST algorithms in a black-box fashion.

2.1.2 Round compression

First introduced by CŁMMOSP [17], the goal of round compression is to simulate multiple rounds of an iterative algorithm within a single MPC round. To do so, one gathers "sufficient amount of information" into a single machine. For example, if an iterative algorithm $\mathcal{A}$ only needs to know the k-hop neighborhood to perform r steps of an algorithm, then these r steps can be compressed into a single MPC round once the k-hop neighborhood has been gathered.

2.1.3 Graph exponentiation

One way to speed up computation in an all-to-all communication setting (such as MPC) is the well-known graph exponentiation technique of Lenzen and Wattenhofer [31]. The idea is as follows: Suppose each vertex is currently aware of its 2^{k-1}-hop neighborhood, then by sending this 2^{k-1} topology to all their current neighbors, each vertex learns about their respective 2^k-hop neighborhoods in one additional MPC round. In other words, every vertex can learn about its k-hop neighborhood in $\mathcal{O}(\log k)$ MPC rounds, as long as the machine memory is large enough. See Figure 1 for an illustration. This technique is motivated by the fact that once a vertex has gathered its k-hop neighborhood, it can execute any LOCAL algorithm that runs in k rounds in just a single MPC round.

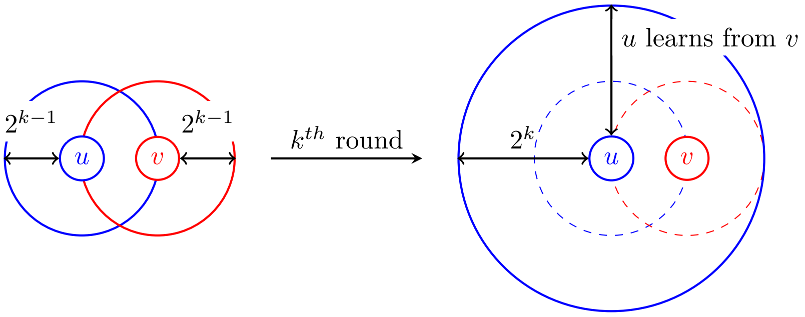


Figure 1 After round k, vertex u knows the graph topology within its 2^k-hop neighborhood.

2.1.4 Combining graph exponentiation with round compression

Suppose we wish to execute a k-round LOCAL algorithm but the machine memory of a single machine is too small to contain entire k-hop neighborhoods. To get around this, one can combine graph exponentiation with round compression:

1. All vertices collect the largest possible neighborhood using graph exponentiation.
2. Suppose ℓ-hop neighborhoods were collected, for some $\ell < k$. All vertices simulate ℓ steps of the LOCAL algorithm in a single MPC round using round compression.
3. All vertices update their neighbors about the status of their computation.
4. Repeat steps 2-3 for $\mathcal{O}(k/\ell)$ phases.

This essentially creates a *virtual communication graph* where vertices are connected to their ℓ-hop neighborhoods. This allows a vertex to derive, in one round of MPC, all the messages that reaches it in the next ℓ rounds of message passing. Using one more MPC round and the fact that local computation is unbounded, a vertex can inform all its neighbors in the virtual graph about its current state in the simulated message passing algorithm. See Figure 2.

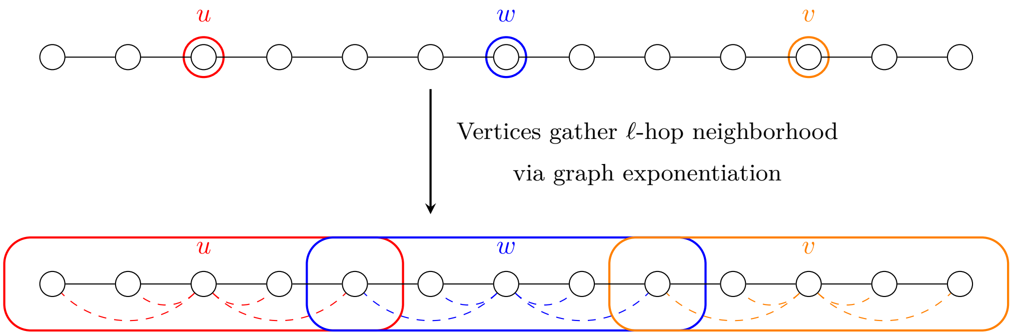


Figure 2 Suppose $\ell = 2$. After each vertex collects their ℓ-hop neighborhood, computation within each collected neighborhood can be performed in a single compressed MPC round. While the vertices u and v were originally 8 hops apart, they can communicate in 2 MPC rounds through vertex w's collected neighborhood in the *virtual communication graph*. Observe that this virtual communication graph has a smaller effective diameter compared to the original input graph.

▶ Remark 4. In Section 2.1.4, we make the implicit assumption that the states of the vertices are small and hence can be communicated with small messages. In many algorithms (e.g. for solving MIS, matching, coloring), including ours, the vertices maintain very small states. Hence, we omit discussion of individual message sizes in the scope of this paper.

2.1.5 Broadcast / Convergecast trees

Broadcast trees are a useful MPC data structure introduced by Goodrich, Sitchinava and Zhang [27] that allow us to perform certain aggregation tasks in $\mathcal{O}(1/\delta)$ MPC rounds, which is essentially $\mathcal{O}(1)$ for constant δ. Suppose we have $\mathcal{O}(N)$ global memory and $S = \mathcal{O}\left(n^{\delta}\right)$ local memory.[11] We build an S-ary virtual communication tree over the machines. That is, within one MPC round, the parent machine can send $\mathcal{O}(1)$ numbers to each of its S children machines, or collect one number from each of its S children machines. In $\mathcal{O}(\log_S N) \subseteq \mathcal{O}(1/\delta)$ rounds, for all vertices v in parallel, one can:

- broadcast a message from v to all neighboring vertices in $N(v)$;
- compute $f(N(v))$, the value of a distributive aggregate function f on set of vertices $N(v)$.

An example of such a function f is computing the sum/min/max of numbers that were originally distributed across all machines. We use broadcast trees in the MPC implementation of the algorithm described in Corollary 32.

2.2 Randomized greedy MIS on bounded degree graphs

The following result of Fischer and Noever [21] states that each vertex only needs the ordering of the vertices within its $\mathcal{O}(\log n)$-hop neighborhood in order to compute its own output status within a randomized greedy MIS run.[12]

▶ **Theorem 5** (Fischer and Noever [21]). *Given a uniform-at-random ordering of vertices, with high probability, the MIS status of any vertex is determined by the vertex orderings within its $\mathcal{O}(\log n)$-hop neighborhood.*

[11] We borrow some notation from Ghaffari and Nowicki [25, Lemma 3.5]. For n-vertex graphs, $N \in \mathcal{O}(n^2)$.

[12] More specifically, they analyzed the "longest length of a dependency path" and showed that it is $\mathcal{O}(\log n)$ with high probability, which implies Theorem 5.

Let $\pi : [n] \to V$ be a uniform-at-random ordering of vertices and G be a graph with maximum degree Δ. In Section 3, we show that one can compute greedy MIS (with respect to π) in $\mathcal{O}(\log \Delta \cdot \log \log n)$ MPC rounds.

▶ **Theorem 6** (Randomized greedy MIS (Informal))**.** *Let G be a graph with maximum degree Δ. Then, randomized greedy MIS can be computed in $\mathcal{O}\left(\log \Delta \cdot \log^3 \log n\right)$ MPC rounds in Model 1, or in $\mathcal{O}(\log \Delta \cdot \log \log n)$ MPC rounds in Model 2.*

Algorithm 1 works in phases. In each phase, we process a prefix graph G_{prefix} defined by vertices indexed by a prefix of π, where the maximum degree is $\mathcal{O}(\log n)$ by Chernoff bounds. Algorithm 2 and Algorithm 3 are two subroutines to process prefix graph G_{prefix}. The latter subroutine is faster by a $\log^2 \log n$ factor but assumes access to more machines. For a sufficiently large prefix of π, the maximum degree of the input graph after processing G_{prefix} drops to $\Delta/2$ with high probability. This concludes a phase. Since the maximum degree in the original graph is halved, we can process more vertices in subsequent phases and thus process all n vertices after $\mathcal{O}(\log \Delta)$ phases. See Figure 3 for an illustration.

▶ Remark 7 (Discussion about maximum degree). We implicitly assume that $\Delta > 1$, which can be checked in $\mathcal{O}(1)$ rounds. Otherwise, when $\Delta = 1$, the graph only contain pairs of vertices and isolated vertices and greedy MIS can be trivially simulated in one round.

Algorithm 1 Greedy MIS in sublinear memory regime of the MPC model.

```
Input: Graph $G = (V, E)$ with maximum degree $\Delta$
Let $\pi : [n] \to V$ be an ordering of vertices chosen uniformly at random.
for $i = 0, 1, 2, \ldots, \mathcal{O}(\log \Delta)$ do        ▷ $\mathcal{O}(\log \Delta)$ phases, or until $G$ is empty
    Let prefix size $t_i = \mathcal{O}\left(\frac{n \log n}{\Delta/2^i}\right)$ and prefix offset $o_i = \sum_{z=0}^{i-1} t_z$.
    $G_i \leftarrow$ Prefix graph induced by vertices $\pi(o_i + 1), \ldots, \pi(o_i + t_i)$ with max. degree $\Delta'$.
    Process $G_i$ using Algorithm 2 or Algorithm 3. ▷ By Chernoff bounds, $\Delta' \in \mathcal{O}(\log n)$
end for
Process any remaining vertices in $G$ using additional $\mathcal{O}(\log \log n)$ MPC rounds.
```

Algorithm 2 Greedy MIS on n-vertex graph in $\mathcal{O}\left(\log^2 \Delta \cdot \log \log n\right)$ MPC rounds in Model 1.

```
Input: Vertex ordering $\pi$, graph $G$ on $n$ vertices with maximum degree $\Delta$
for $i = 0, 1, 2, \ldots, \lceil \log_2 \Delta \rceil$ do        ▷ $\mathcal{O}(\log \Delta)$ phases, or until $G$ is empty
    Let chunk size $c_i = \frac{2^i}{100\Delta} \cdot n$.        ▷ Chunk size doubles per phase
    for $j = 1, 2, \ldots, 2000 \log \Delta$ do        ▷ $\mathcal{O}(\log \Delta)$ iterations, or until $G$ is empty
        Let offset $o_{i,j} = c_i \cdot (j - 1) + \sum_{z=0}^{i-1} c_z \cdot 2000 \log \Delta$.
        Let chunk graph $G_{i,j}$ be the graph induced by vertices $\pi(o_{i,j}), \ldots, \pi(o_{i,j} + c_i)$.
        Process chunk graph $G_{i,j}$.
    end for
end for
```

▶ Remark 8 (Discussion about Algorithm 2). Our algorithm is inspired by the idea of graph shattering introduced by BEPS [6]. We break up the simulation of greedy MIS on G_{prefix} into $\mathcal{O}(\log \Delta)$ phases that process chunks of increasing size within the prefix graph. By performing $\mathcal{O}(\log \Delta)$ iterations within a phase, we can argue that any vertex in the remaining prefix graph has "low degree" with high probability in Δ. This allows us to prove that the connected components while processing every chunk of vertices is at most $\mathcal{O}(\log n)$ and each vertex can learn (within the global memory limits) about the topology of its connected component in $\mathcal{O}(\log \log n)$ rounds via graph exponentiation. The constants 100 and 2000 are chosen for a cleaner analysis. While the algorithm indexes more than n vertices, we simply terminate after processing the last vertex in the permutation.[13]

Algorithm 3 Greedy MIS on n-vertex graph in $\mathcal{O}(\log \log n + \log \Delta)$ MPC rounds in Model 2.

```
Input: Vertex ordering $\pi$, graph $G$ on $n$ vertices with maximum degree $\Delta$
Assign a machine to each vertex.        ▷ In Model 2, we have $\geq n$ machines.
Graph exponentiate and gather $R$-hop neighborhood, where $R \in \mathcal{O}\left(\frac{\log n}{\log \Delta}\right)$.
Simulate greedy MIS (with respect to $\pi$) in $\mathcal{O}(\log \Delta)$ MPC rounds.
```

▶ Remark 9 (Discussion about Algorithm 3). We know from Theorem 5 that it suffices for each vertex know its $\mathcal{O}(\log n)$-hop neighborhood in order to determine whether it is in the greedy MIS. However, the $\mathcal{O}(\log n)$-hop neighborhoods may not fit in a single machine. So, we use Section 2.1.4 to obtain a running time of $\mathcal{O}\left(\log R + \frac{\log n}{R}\right) \subseteq \mathcal{O}(\log \log n + \log \Delta)$.

▶ Remark 10 (Comparison with the work of Blelloch, Fineman and Shun (BFS) [10]). The algorithm of BFS [10] also considered prefixes of increasing size and they have a similar lemma as our Lemma 22. However, their work does not immediately imply ours. The focus of BFS [10] was in the PRAM model in which the goal is to obtain an algorithm that is small work-depth – they gave implementation of their algorithms that does a linear amount of work with polylogarithmic depth. In this work, we are interested in studying the MPC model, in particular the sublinear memory regime. Directly simulating their algorithm in MPC yields an algorithm that runs in $\mathcal{O}(\log \Delta \cdot \log n)$ rounds. Here, we crucially exploit graph exponentiation and round compression to speed up the greedy MIS simulation prefix graphs, enabling us to obtain algorithms that have an exponentially better dependency on n, i.e. that run in $\mathcal{O}(\log \Delta \cdot \operatorname{poly}(\log \log n))$ rounds.

2.3 Correlation clustering on bounded arboricity graphs

Our algorithmic results for correlation clustering derive from the following key structural lemma that is proven by arguing that a local improvement to the clustering cost is possible if there exists large clusters.

▶ **Lemma 11** (Structural lemma for correlation clustering (Informal)). *There exists an optimum correlation clustering where all clusters have size at most* $4\lambda - 2$.

This structural lemma allows us to perform cost-charging arguments against *some* optimum clustering with bounded cluster sizes. In particular, if a vertex has degree much larger than λ, then many of its incident edges incur disagreements. This insight yields the following algorithmic implication: we can effectively ignore high-degree vertices.

▶ **Theorem 12** (Algorithmic implication (Informal)). *Let G be a graph where E^+ induces a λ-arboric graph. Form singleton clusters with vertices with degrees $\mathcal{O}(\lambda/\varepsilon)$. Run an α-approximate algorithm $\mathcal{A}$ on the remaining subgraph. Then, the union of clusters is a* $\max\{1+\varepsilon, \alpha\}$*-approximation. The runtime and approximation guarantees of the overall algorithm follows from the guarantees of $\mathcal{A}$ (e.g. in expectation / worst case, det. / rand.).*

[13] After all phases, we would have processed $\sum_{j=0}^{\lceil \log_2 \Delta \rceil} c_j \cdot 2000 \log \Delta \geq \frac{2^{\lceil \log_2 \Delta \rceil}}{\Delta} \cdot n \cdot 2000 \log \Delta \geq n$ vertices.

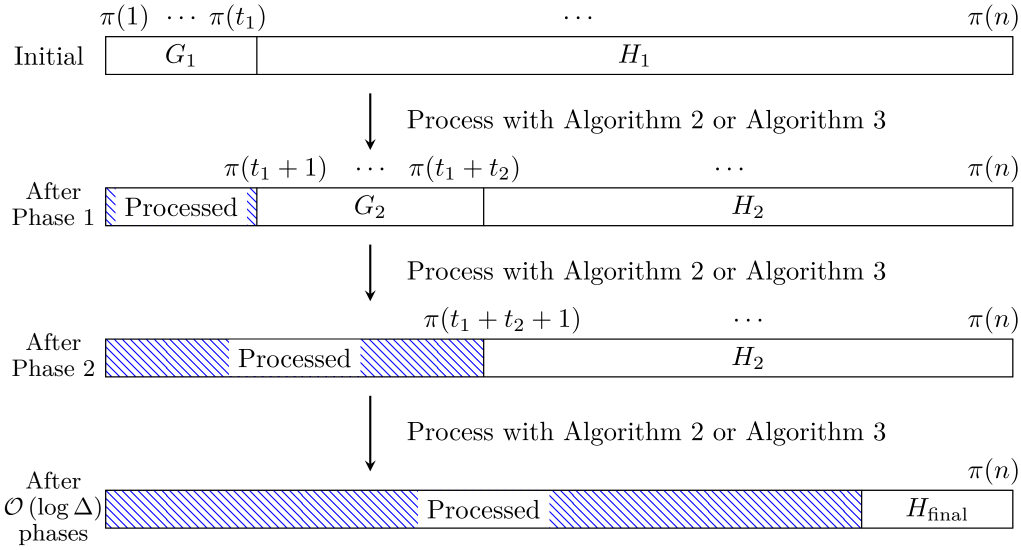

$\pi(1)$ $\cdots$ $\pi(t_1)$
$\cdots$
$\pi(n)$
Initial
G_1
H_1
Process with Algorithm 2 or Algorithm 3
$\pi(t_1+1)$ $\cdots$ $\pi(t_1+t_2)$
$\cdots$
$\pi(n)$
After Phase 1
Processed
G_2
H_2
Process with Algorithm 2 or Algorithm 3
$\pi(t_1+t_2+1)$
$\cdots$
$\pi(n)$
After Phase 2
Processed
H_2
Process with Algorithm 2 or Algorithm 3
$\pi(n)$
After $\mathcal{O}(\log \Delta)$ phases
Processed
H_{final}

Figure 3 Illustration of Algorithm 1 given an initial graph G on n vertices with maximum degree Δ. Let $i \in \{1, \dots, \mathcal{O}(\log \Delta)\}$ and define $t_i = \mathcal{O}\left(\frac{n \log n}{\Delta/2^i}\right)$. For each i, with high probability, the induced subgraph G_i has maximum degree poly$(\log n)$. To process G_i, apply Algorithm 2 in $\mathcal{O}(\log^3 \log n)$ MPC rounds, or Algorithm 3 in $\mathcal{O}(\log \log n)$ MPC rounds while using extra global memory. By our choice of t_i, Lemma 22 tells us that remaining subgraph H_i has maximum degree $\Delta/2^i$. We repeat this argument until the final subgraph H_{final} involving poly$(\log n)$ vertices, which can be processed in another call to Algorithm 2 or Algorithm 3.

Observe that PIVOT essentially simulates a randomized greedy MIS with respect to a uniform-at-random ordering of vertices. By setting $\varepsilon = 2$ in Theorem 12 and $\Delta = \mathcal{O}(\lambda)$ in Theorem 6, we immediately obtain a 3-approximation (in expectation) algorithm for correlation clustering in $\mathcal{O}(\log \lambda \cdot \text{poly}(\log \log n))$ MPC rounds. Note that we always have $\lambda \leq \Delta \leq n$, and that λ can be significantly smaller than Δ and n in general. Many sparse graphs have $\lambda \in \mathcal{O}(1)$ while having unbounded maximum degrees, including planar graphs and bounded treewidth graphs. As such, for several classes of graphs, our result improves over directly simulating PIVOT in $\mathcal{O}(\log n)$ rounds.

▶ **Corollary 13** (General algorithm (Informal)). *Let G be a complete signed graph such that E^+ induces a λ-arboric graph. There exists an algorithm that, with high probability, produces a 3-approximation (in expectation) for correlation clustering of G in $\mathcal{O}\left(\log \lambda \cdot \log^3 \log n\right)$ MPC rounds in Model 1, or $\mathcal{O}\left(\log \lambda \cdot \log \log n\right)$ MPC rounds in Model 2.*

▶ Remark 14 (On converting "in expectation" to "with high probability"). Note that one can run $\mathcal{O}(\log n)$ copies of Corollary 13 in parallel and output the best clustering. Applying this standard trick converts the "in expectation" guarantee to a "with high probability" guarantee with only a logarithmic factor increase in memory consumption.

For forests with $\lambda = 1$, Lemma 11 states that the optimum correlation clustering cost corresponds to the number of edges minus the size of the maximum matching. Instead of computing a maximum matching, Lemma 15 tells us that using an approximate matching suffices to obtain an α-approximation (not necessarily maximal) to the correlation clustering problem. Note that maximal matchings are 2-approximations and they always apply.

▶ **Lemma 15** (Approximation via approximate matchings (Informal)). *Let G be a complete signed graph such that E^+ induces a forest. Suppose that the maximum matching size on E^+ is $|M^*|$. If M is a matching on E^+ such that $\alpha \cdot |M| \geq |M^*|$, for some $1 \leq \alpha \leq 2$, then clustering using M yields an α-approximation to the optimum correlation clustering of G.*

Thus, it suffices to apply known maximum/approximate matching algorithms in sublinear memory regime of MPC to obtain correlation clustering algorithms in the special case of $\lambda = 1$. More specifically, we consider the following results.

- Using dynamic programming, BBDHM [7] compute a maximum matching (on trees) in $\mathcal{O}(\log n)$ MPC rounds.
- In the LOCAL model, EMR [20] deterministically solve $(1+\varepsilon)$-approx. matching in $\mathcal{O}\left(\Delta^{\mathcal{O}\left(\frac{1}{\varepsilon}\right)} + \frac{1}{\varepsilon^2} \cdot \log^* n\right)$ rounds.
- In the CONGEST model, BCGS [5] give an $\mathcal{O}\left(2^{\mathcal{O}(1/\varepsilon)} \cdot \frac{\log \Delta}{\log \log \Delta}\right)$ round randomized algorithm for $(1+\varepsilon)$-approx. matching.

These approximation results are heavily based on the Hopcroft-Karp framework [28], where independent sets of augmenting paths are iteratively flipped. Since $\lambda = 1$ and ε is a constant, we have a subgraph of constant maximum degree by ignoring vertices with degrees $\mathcal{O}(\lambda/\varepsilon)$. On this constant degree graph, each vertex only needs polylogarithmic memory when we perform graph exponentiation, satisfying the memory constraints of Model 1. Applying these matching algorithms together with Theorem 12 and Lemma 15 yields the following result.

▶ **Corollary 16** (Forest algorithm (Informal)). *Let G be a complete signed graph such that E^+ induces a forest and $0 < \varepsilon \leq 1$ be a constant. Hiding factors in $1/\varepsilon$ using $\mathcal{O}_\varepsilon(\cdot)$, there exists:*

1. *An optimum randomized algorithm that runs in $\mathcal{O}(\log n)$ MPC rounds.*
2. *A $(1+\varepsilon)$-approximation (worst case) det. algo. that runs in $\mathcal{O}_\varepsilon(\log \log^* n)$ MPC rounds.*
3. *A $(1+\varepsilon)$-approximation (worst case) randomized algo. that runs in $\mathcal{O}_\varepsilon(1)$ MPC rounds.*

Finally, we give a simple $\mathcal{O}(\lambda^2)$-approximate (worst-case) algorithm in $\mathcal{O}(1)$ MPC rounds.

▶ **Corollary 17** (Simple algorithm (Informal)). *Let G be a complete signed graph such that E^+ induces a λ-arboric graph. Then, there exists an $\mathcal{O}(\lambda^2)$-approximation (worst case) deterministic algorithm that runs in $\mathcal{O}(1)$ MPC rounds.*

The simple algorithm is as follows: connected components which are cliques form clusters, and all other vertices form individual singleton clusters. This can be implemented in $\mathcal{O}(1)$ MPC rounds using broadcast trees. We now give an informal argument when the input graph is a single connected component but not a clique. By Lemma 11, there will be $\geq n/\lambda$ clusters and so the optimal number of disagreements is $\geq n/\lambda$. Meanwhile, the singleton clusters incurs errors on all positive edges, i.e. $\leq \lambda \cdot n$ since E^+ induces a λ-arboric graph. Thus, the worst possible approximation ratio is $\approx \lambda^2$.

3 Randomized greedy MIS on bounded degree graphs

In this section, we explain how to efficiently compute a randomized greedy MIS in the sublinear memory regime of the MPC model. We will first individually analyze Algorithm 2 and Algorithm 3 and then show how to use them as black-box subroutines in Algorithm 1. Both Algorithm 2 and Algorithm 3 rely on the result of Fischer and Noever [21] that it suffices for each vertex to know the π ordering of its $\mathcal{O}(\log n)$-hop neighborhood.

Algorithm 2 is inspired by the graph shattering idea introduced by BEPS [6]. Our analysis follows a similar outline as the analysis of the maximal independent set of BEPS [6] but is significantly simpler as our "vertex sampling process" in each step simply follows the uniform-at-random vertex permutation π: we do not explicitly handle high-degree vertices at the end, but we argue that connected components are still small even if π chooses some of them. The key crux of our analysis is to argue that, for appropriately defined step sizes, the connected components considered are of size $\mathcal{O}(\log n)$.

▶ **Lemma 18.** *Consider Algorithm 2. With high probability in n, the connected components in any chunk graph $G_{i,j}$ have size $\mathcal{O}(\log n)$.*

This allows us to argue that all vertices involved can learn the full topology of their connected components via graph exponentiation in $\mathcal{O}(\log \log n)$ MPC rounds in Model 1.

▶ **Lemma 19.** *Consider Algorithm 2 in Model 1. Fix an arbitrary chunk graph $G_{i,j}$. If connected components in $G_{i,j}$ have size at most poly$(\log n)$, then every vertex can learn the full topology of its connected component in $\mathcal{O}(\log \log n)$ MPC rounds.*

▶ **Lemma 20.** *Consider Algorithm 2 in Model 1. Suppose $G = (V, E)$ has n vertices with maximum degree Δ. Let π be a uniform-at-random ordering of V. Then, with high probability, one can simulate greedy MIS on G (with respect to π) in $\mathcal{O}\left(\log^2 \Delta \cdot \log \log n\right)$ MPC rounds.*

By using at least n machines, Algorithm 3 presents a simpler and faster algorithm for computing greedy MIS compared to Algorithm 2. It exploits computational features of the MPC model such as graph exponentiation and round compression to speed up computation.

▶ **Lemma 21.** *Consider Algorithm 3 in Model 2. Suppose $G = (V, E)$ has n vertices with maximum degree Δ. Let π be a uniform-at-random ordering of V. Then, with high probability, one can simulate greedy MIS on G (with respect to π) in $\mathcal{O}\left(\log \log n + \log \Delta\right)$ MPC rounds.*

Recall that Algorithm 1 uses Algorithm 2 or Algorithm 3 as subroutines to compute the greedy MIS on a subgraph induced by some prefix of π's ordering in each phase. We first prove Lemma 22 which bounds the maximum degree of the remaining subgraph after processing $t \leq n$ vertices. By our choice of prefix sizes, we see that the maximum degree is halved with high probability in each phase and thus $\mathcal{O}(\log \Delta)$ phases suffice.

▶ **Lemma 22.** *Let G be a graph on n vertices and $\pi : [n] \to V$ be a uniform-at-random ordering of vertices. For $t \in [n]$, consider the subgraph H_t obtained after processing vertices $\{\pi(1), \ldots, \pi(t)\}$ via greedy MIS (with respect to π). Then, with high probability, the maximum degree in H_t is at most $\mathcal{O}\left(\frac{n \log n}{t}\right)$.*

▶ Remark 23. Similar statements to Lemma 21 and Lemma 22 were previously known.[14]

▶ **Theorem 24.** *Let G be a graph with n vertices of maximum degree Δ and $\pi : [n] \to V$ be a uniform-at-random ordering of vertices. Then, with high probability, one can compute greedy MIS (with respect to π) in $\mathcal{O}\left(\log \Delta \cdot \log^3 \log n\right)$ MPC rounds in Model 1, or $\mathcal{O}\left(\log \Delta \cdot \log \log n\right)$ MPC rounds in Model 2.*

4 Structural properties for correlation clustering

In this section, we prove our main result (Theorem 26) about correlation clustering by ignoring high-degree vertices. To do so, we first show a structural result of optimum correlation clusterings (Lemma 25): there *exists* an optimum clustering with bounded cluster sizes. This structural lemma also implies that in the special case of forests (i.e. $\lambda = 1$), a maximum matching on E^+ yields an optimum correlation clustering of G (Corollary 27).

[14] E.g. see GGKMR [23, Section 3], ACGMW [1, Lemma 27], and BFS [10, Lemma 3.1].

▶ **Lemma 25** (Structural lemma for correlation clustering). *Let G be a complete signed graph such that positive edges E^+ induce a λ-arboric graph. Then, there exists an optimum correlation clustering where all clusters have size at most $4\lambda - 2$.*

Proof sketch. The proof involves performing local updates by repeatedly removing vertices from large clusters while arguing that the number of disagreements does not increase (it may not strictly decrease but may stay the same). ◀

▶ **Theorem 26** (Algorithmic implication of Lemma 25). *Let G be a complete signed graph such that positive edges E^+ induce a λ-arboric graph. For $\varepsilon > 0$, let*

$$H = \left\{ v \in V : d(v) > \frac{8(1+\varepsilon)}{\varepsilon} \cdot \lambda \right\} \subseteq V$$

be the set of high-degree vertices, and $G' \subseteq G$ be the subgraph obtained by removing high-degree vertices in H. Suppose $\mathcal{A}$ is an α-approximate correlation clustering algorithm and $cost(OPT(G))$ is the optimum correlation clustering cost. Then,

$$cost\left(\{\{v\} : v \in H\} \cup \mathcal{A}(G')\right) \leq \max\left\{1+\varepsilon, \alpha\right\} \cdot cost(OPT(G))$$

where $\{\{v\} : v \in H\} \cup \mathcal{A}(G')$ is the clustering obtained by combining the singleton clusters of high-degree vertices with $\mathcal{A}$'s clustering of G'. See Algorithm 4 for a pseudocode. Furthermore, if $\mathcal{A}$ is α-approximation only in expectation, then the above inequality holds only in expectation.

Proof sketch. Fix an optimum clustering $OPT(G)$ of G where each cluster has size at most $4\lambda - 2$. Such a clustering exists by Lemma 25. One can then show that $cost(OPT(G)) \geq \frac{1}{1+\varepsilon} \cdot |M^+| + (\text{disagreements in } U)$, where $|M^+|$ is the number of positive edges adjacent to high-degree vertices and U is the set of edges *not* adjacent to any high-degree vertex. The result follows by combining singleton clusters of high-degree vertices H and the α-approximate clustering on low-degree vertices U using $\mathcal{A}$. ◀

Algorithm 4 Correlation clustering for G such that E^+ induces a λ-arboric graph.

```
Input: Graph $G$, $\varepsilon > 0$, $\alpha$-approximate algorithm $\mathcal{A}$
Let $H = \left\{ v \in V : d(v) > \frac{8(1+\varepsilon)}{\varepsilon} \cdot \lambda \right\} \subseteq V$ be the set of high-degree vertices.
Let $G' \subseteq G$ be a bounded degree subgraph obtained by removing high-degree vertices $H$.
Let $\mathcal{A}(G')$ be the clustering obtained by running $\mathcal{A}$ on the subgraph $G'$.
Return Clustering $\{\{v\} : v \in H\} \cup \mathcal{A}(G')$.
```

▶ **Corollary 27** (Maximum matchings yield optimum correlation clustering in forests). *Let G be a complete signed graph such that positive edges E^+ induce a forest (i.e. $\lambda = 1$). Then, clustering using a* maximum *matching on E^+ yields an optimum cost correlation clustering.*

5 Minimizing disagreements in bounded arboricity graphs and forests

We now describe how to use our main result (Theorem 26) to obtain efficient correlation clustering algorithms in the sublinear memory regime of the MPC model. Theorem 26 implies that we can focus on solving correlation clustering on graphs with maximum degree $\mathcal{O}(\lambda)$.

For general λ-arboric graphs, we simulate `PIVOT` by invoking Theorem 24 to obtain Corollary 28. For forests, Corollary 27 states that a maximum matching on E^+ yields an optimal correlation clustering. Then, Lemma 29 tells us that if one computes an approximate matching (not necessarily maximal) instead of a maximum matching, we still get a reasonable cost approximation to the optimum correlation clustering. By invoking existing matching algorithms, we show how to obtain three different correlation clustering algorithms (with different guarantees) in Corollary 31. Finally, Corollary 32 gives a deterministic constant round algorithm that yields an $\mathcal{O}(\lambda^2)$ approximation.

▶ **Corollary 28.** *Let G be a complete signed graph such that positive edges E^+ induce a λ-arboric graph. With high probability, there exists an algorithm that produces a 3-approximation (in expectation) for correlation clustering of G in $\mathcal{O}\left(\log \lambda \cdot \log^3 \log n\right)$ MPC rounds in Model 1, or $\mathcal{O}\left(\log \lambda \cdot \log \log n\right)$ MPC rounds in Model 2.*

▶ **Lemma 29.** *Let G be a complete signed graph such that positive edges E^+ induce a forest. Suppose $|M^*|$ is the size of a maximum matching on E^+ and M is an approximate matching on E^+ where $\alpha \cdot |M| \geq |M^*|$ for some $1 \leq \alpha \leq 2$. Then, clustering using M yields an α-approximation to the optimum correlation clustering of G.*

▶ Remark 30. The approximation ratio of Lemma 29 tends to 1 as $|M|$ tends to $|M^*|$. The worst ratio possible is 2 and this approximation ratio is tight: consider a path of 4 vertices and 3 edges with $|M^*| = 2$ and maximal matching $|M| = 1$.

▶ **Corollary 31.** *Consider Model 1. Let G be a complete signed graph such that positive edges E^+ induce a forest. Let $0 < \varepsilon \leq 1$ be a constant. Then, there exists the following algorithms for correlation clustering:*

1. *An optimum randomized algorithm that runs in $\widetilde{\mathcal{O}}(\log n)$ MPC rounds.*
2. *A $(1+\varepsilon)$-approx. (worst case) deterministic algo. that runs in $\mathcal{O}\left(\frac{1}{\varepsilon} \cdot \left(\log \frac{1}{\varepsilon} + \log \log^* n\right)\right)$ MPC rounds.*
3. *A $(1+\varepsilon)$-approx. (worst case) randomized algo. that runs in $\mathcal{O}\left(\log \log \frac{1}{\varepsilon}\right)$ MPC rounds.*

▶ **Corollary 32.** *Consider Model 1. Let G be a complete signed graph such that positive edges E^+ induce a λ-arboric graph. Then, there exists a deterministic algorithm that produces an $\mathcal{O}(\lambda^2)$-approximation (worst case) for correlation clustering of G in $\mathcal{O}(1)$ MPC rounds.*

▶ Remark 33. The approximation analysis in Corollary 32 is tight (up to constant factors): consider the barbell graph where two cliques K_λ (cliques on λ vertices) are joined by a single edge. The optimum clustering forms a cluster on each K_λ and incurs one external disagreement. Meanwhile, forming singleton clusters incurs $\approx \lambda^2$ positive disagreements.

6 Conclusions and open questions

In this work, we present a structural result on correlation clustering of complete signed graphs such that the positive edges induce a bounded arboricity graph. Combining this with known algorithms, we obtain efficient algorithms in the sublinear memory regime of the MPC model. We also showed how to compute a *randomized greedy MIS* in $\mathcal{O}\left(\log \Delta \cdot \log \log n\right)$ MPC rounds. As intriguing directions for future work, we pose the following questions:

▶ **Question 1.** *For graphs with maximum degree $\Delta \in poly(\log n)$, can one compute greedy MIS in $\mathcal{O}\left(\log \log n\right)$ MPC rounds in the sublinear memory regime of the MPC model?*

For graphs with maximum degree $\Delta \in \text{poly}(\log n)$, Algorithm 2 runs in $\mathcal{O}\left(\log^3 \log n\right)$ MPC rounds and Algorithm 3 runs in $\mathcal{O}(\log \log n)$ MPC rounds assuming access to at least n machines. Is it possible to achieve running time of $\mathcal{O}(\log \log n)$ MPC rounds without additional global memory assumptions?

▶ **Question 2.** *Can a randomized greedy MIS be computed in $\mathcal{O}(\log \Delta + \log \log n)$ or $\mathcal{O}\left(\sqrt{\log \Delta} + \log \log n\right)$ MPC rounds?*

This would imply that a 3-approximate (in expectation) correlation clustering algorithm in the same number of MPC rounds. We posit that a better running time than $\mathcal{O}(\log \Delta \cdot \log \log n)$ should be possible. The informal intuition is as follows: Fischer and Noever's result [21] tells us that most vertices do not have long dependency chains in every phase, so "pipelining arguments" might work.

▶ **Question 3.** *Is there an efficient* distributed *algorithm to minimize disagreements with an approximation guarantee strictly better than 3 (in expectation), or worst-case guarantees for general graphs?*

For minimizing disagreements in complete signed graphs, known algorithms (see Section 1.4) with approximation guarantees strictly less than 3 (in expectation) are based on probabilistic rounding of LPs. *Can one implement such LPs efficiently in a distributed setting, or design an algorithm that is amenable to a distributed implementation with provable guarantees strictly better than 3?* In this work, we gave algorithms with worst-case approximation guarantees when the graph induced by positive edges is a forest. *Can one design algorithms that give worst-case guarantees for general graphs?*

References

1 Kook Jin Ahn, Graham Cormode, Sudipto Guha, Andrew McGregor, and Anthony Wirth. Correlation Clustering in Data Streams. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML)*, pages 2237–2246, 2015.

2 Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating Inconsistent Information: Ranking and Clustering. *Journal of the ACM (JACM)*, 55(5):1–27, 2008.

3 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 674–685. IEEE, 2018.

4 Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation Clustering. *Machine learning*, 56(1-3):89–113, 2004.

5 Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed Approximation of Maximum Independent Set and Maximum Matching. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 165–174, 2017.

6 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The Locality of Distributed Symmetry Breaking. *Journal of the ACM (JACM)*, 63(3):1–45, 2016.

7 MohammadHossein Bateni, Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, and Vahab Mirrokni. Brief Announcement: MapReduce Algorithms for Massive Trees. In *45th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 162:1–162:4, 2018.

8 Paul Beame, Paraschos Koutris, and Dan Suciu. Communication Steps for Parallel Query Processing. *Journal of the ACM (JACM)*, 64(6):1–58, 2017.

9 Soheil Behnezhad, Sebastian Brandt, Mahsa Derakhshan, Manuela Fischer, MohammadTaghi Hajiaghayi, Richard M. Karp, and Jara Uitto. Massively Parallel Computation of Matching and MIS in Sparse Graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 481–490, 2019.

10 Guy Blelloch, Jeremy Fineman, and Julian Shun. Greedy Sequential Maximal Independent Set and Matching Are Parallel on Average. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 308—-317, 2012.

11 Nicolò Cesa-Bianchi, Claudio Gentile, Fabio Vitale, and Giovanni Zappella. A Correlation Clustering Approach to Link Classification in Signed Networks. *Journal of Machine Learning Research (JMLR)*, 23:34.1–34.20, 2013.

12 Moses Charikar, Venkatesan Guruswami, and Anthony Wirth. Clustering with Qualitative Information. *Journal of Computer and System Sciences*, 71(3):360–383, 2005.

13 Shuchi Chawla, Konstantin Makarychev, Tselil Schramm, and Grigory Yaroslavtsev. Near Optimal LP Rounding Algorithm for Correlation Clustering on Complete and Complete k-partite graphs. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing (STOC)*, pages 219–228, 2015.

14 Yudong Chen, Sujay Sanghavi, and Huan Xu. Clustering Sparse Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2204–2212, 2012.

15 Flavio Chierichetti, Nilesh Dalvi, and Ravi Kumar. Correlation Clustering in MapReduce. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge Discovery and Data mining (KDD)*, pages 641–650, 2014.

16 Artur Czumaj, Peter Davies, and Merav Parter. Graph sparsification for derandomizing massively parallel computation with low space. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 175–185, 2020.

17 Artur Czumaj, Jakub Łacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round Compression for Parallel Matching Algorithms. *SIAM Journal on Computing*, 49(5):STOC18–1, 2019.

18 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

19 Erik D Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. Correlation Clustering in General Weighted Graphs. *Theoretical Computer Science*, 361(2-3):172–187, 2006.

20 Guy Even, Moti Medina, and Dana Ron. Distributed Maximum Matching in Bounded Degree Graphs. In *Proceedings of International Conference on Distributed Computing and Networking (ICDCN)*, 2015.

21 Manuela Fischer and Andreas Noever. Tight Analysis of Parallel Randomized Greedy MIS. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2152–2160, 2018.

22 Mohsen Ghaffari. Massively Parallel Algorithms, 2019. Available at: `https://people.inf.ethz.ch/gmohsen/MPA19/Notes/MPA.pdf`.

23 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2018.

24 Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved MPC Algorithms for MIS, Matching, and Coloring on Trees and Beyond. In *34th International Symposium on Distributed Computing (DISC)*, pages 34:1–34:18, 2020.

25 Mohsen Ghaffari and Krzysztof Nowicki. Massively Parallel Algorithms for Minimum Cut. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC)*, pages 119–128, 2020.

26 Mohsen Ghaffari and Jara Uitto. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1636–1653, 2019.

27 Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 374–383, 2011.

28 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

29 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

30 Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the 21st annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

31 Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential Speed-up of Local Algorithms Using Non-local Communication. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, pages 295–296, 2010.

32 Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on computing*, 21(1):193–201, 1992.

33 Xinghao Pan, Dimitris Papailiopoulos, Samet Oymak, Benjamin Recht, Kannan Ramchandran, and Michael I Jordan. Parallel Correlation Clustering on Big Graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 82–90, 2015.

34 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

35 Chaitanya Swamy. Correlation Clustering: Maximizing Agreements via Semidefinite Programming. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 4, pages 526–527, 2004.

36 Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

37 Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.

Fully Read/Write Fence-Free Work-Stealing with Multiplicity

Armando Castañeda
Institute of Mathematics, National Autonomous University of Mexico, Mexico City, Mexico

Miguel Piña
Faculty of Sciences, National Autonomous University of Mexico, Mexico City, Mexico

Abstract

It is known that any algorithm for *work-stealing* in the standard asynchronous shared memory model must use expensive Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions. There have been proposed algorithms for relaxations in the standard model and algorithms in restricted models that avoid the impossibility result, but only in some operations.

This paper considers work-stealing with *multiplicity*, a relaxation in which every task is taken by *at least* one operation, with the requirement that any process can extract a task *at most once*. Two versions of the relaxation are considered and two *fully* Read/Write algorithms are presented in the standard asynchronous shared memory model, both devoid of Read-After-Write synchronization patterns in all its operations, the second algorithm additionally being *fully fence-free*, namely, no specific ordering among the algorithm's instructions is required, beyond what is implied by data dependence. To our knowledge, these are the first algorithms for work-stealing possessing all these properties. Our algorithms are also wait-free solutions of relaxed versions of single-enqueue multi-dequeuer queues. The algorithms are obtained by reducing work-stealing with multiplicity and weak multiplicity to MaxRegister and RangeMaxRegister, a relaxation of MaxRegister which might be of independent interest. An experimental evaluation shows that our fully fence-free algorithm exhibits better performance than Cilk THE, Chase-Lev and Idempotent Work-Stealing algorithms.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Computing methodologies → Distributed algorithms; Computing methodologies → Concurrent algorithms; Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Correctness condition, Linearizability, Nonblocking, Relaxed data type, Set-linearizability, Wait-freedom, Work-stealing

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.16

Related Version *Full Version*: https://arxiv.org/abs/2008.04424 [8]

Supplementary Material *Software (Source Code)*: https://bit.ly/2WHomWg
archived at swh:1:dir:a6173150e85d59f84d9840aff7f2bc92d194f858

Funding *Armando Castañeda*: Supported by UNAM-PAPIIT project IN108720.
Miguel Piña: Recipient of a PhD fellowship from CONACyT.

1 Introduction

Context. *Work-stealing* is a popular technique to implement dynamic *load balancing* for efficient task parallelization of irregular workloads. It has been used in several contexts, from programming languages and parallel-programming frameworks to SAT solver and state space search exploration in model checking (e.g. [5, 7, 11, 14, 15, 24, 28]). In work-stealing, each process *owns* a set of tasks that have to be executed. The *owner* of the set can put tasks in it and can take tasks from it to execute them. When a process runs out of tasks (i.e. the set is empty), instead of being idle, it becomes a *thief* to steal tasks from a *victim*. Thus, a work-stealing algorithm provides three high-level operations: Put and Take, which can be invoked only by the owner, and Steal, which can be invoked by any thief.

A main target when designing work-stealing algorithms is to make Put and Take as simple and efficient as possible. Unfortunately, it has been shown that any work-stealing algorithm in the standard asynchronous shared memory model must use Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions (e.g. Compare&Swap or Test&Set) [4]. Read-After-Write synchronization patterns are based on the *flag principle* [20] (i.e. writing on a shared variable and then reading another variable), and hence when implementing an algorithm using such a synchronization pattern in real multicore architectures, a memory *fence* (also called *barrier*) is required so that the read and write instructions are not reordered by the compiler or the processor. It is well-known that fences that avoid reads and writes to be reordered are highly costly, while atomic Read-Modify-Write instructions, with high coordination power (which can be formally measured through the *consensus number* formalism [19]), are in principle slower than the simple Read/Write instructions.[1] Indeed, the known work-stealing algorithms in the literature are based on the flag principle in their Take/Steal operations [12, 15, 16, 17]. Thus, a way to circumvent the result in [4] is to consider work-stealing with relaxed semantics, or to make extra assumptions on the model. As far as we know, [25] and [26] are the only works that have followed these directions.

Observing that in some contexts it is ensured that no task is repeated (e.g. by checking first whether a task is completed) or the nature of the problem solved tolerates repeatable work (e.g. parallel SAT solvers), Michael, Vechev and Saraswat propose *idempotent* work-stealing [25], a relaxation allowing a task to be taken *at least once*, instead of *exactly once*. Three idempotent work-stealing algorithms are presented in [25], that insert/extract tasks in different orders. The relaxation allows each of the algorithms to circumvent the impossibility result in [4], only in its Put and Take operations as they use only Read/Write instructions and are devoid of Read-After-Write synchronization patterns; however, Steal uses Compare&Swap. Moreover, Put requires that some Write instructions are not reordered and Steal requires that some Read instructions are not reordered, and thus fences are required when the algorithms are implemented; fences between Read (resp. Write) instructions, however, are usually not too costly in practice. As for progress guarantees, Put and Take are *wait-free* while Steal is only *nonblocking*.

Morrison and Afek consider the restricted TSO model [29] and present two work-stealing algorithms in [26] whose Put operation is wait-free and uses only Read/Write instructions, and Take and Steal are either nonblocking and use Compare&Swap, or blocking and use a *lock*. The algorithms are clever adaptations of the well-known Cilk THE and Chase-Lev work-stealing algorithms [12, 15] to the TSO model. Generally speaking, in this model Write (resp. Read) instructions cannot be reordered, hence fences among Write (resp. Read) instructions are not needed; additionally, each process has a local buffer where its Write instructions are stored until they are eventually propagated to the main memory (in FIFO order). Reordering some Write (resp. Read) instructions of the algorithms in [26] compromises correctness, however TSO prevents this to happen. To avoid Read-After-Write patterns, it is considered in [26] that buffers are of bounded size.

Contributions. In this paper we are interested in the following theoretical question: whether there are meaningful relaxations of work-stealing that allow us to design fully Read/Write, fully wait-free and fully *fence-free* algorithms in the standard asynchronous shared memory model; fence-free means that algorithm's correctness does not require any specific instruction

[1] In practice, contention might be the dominant factor, namely, an uncontended Read-Modify-Write instruction can be faster than contended Read/Write instructions.

ordering, beyond what is implied by data dependence. In other words, we are interested in knowing if simple synchronization mechanisms are suffice to solve non-trivial relaxations of work-stealing. This question has a practical motivation too, as [25] and [26] have shown that performance of work-stealing implementations can be increased by reducing the usage of Read-After-Write patterns and Read-Modify-Write instructions.

We consider work-stealing with *multiplicity* [10], a relaxation in which every task is taken by *at least* one operation, and, differently from idempotent work-stealing, it requires that if a task is taken by several Take/Steal operations, then they must be *pairwise concurrent*; therefore, no more than the number of processes in the system can take the same task. In our relaxation, tasks are inserted/extracted in FIFO order. We present a fully Read/Write algorithm for work-stealing with multiplicity, with its Put operation being fence-free and its Take and Steal operations being devoid of Read-After-Write synchronization patterns. As for progress, all operations are wait-free with Put having constant *step complexity* and Take and Steal having logarithmic step complexity. The algorithm stands for its simplicity, based on a single instance of a MaxRegister object [3, 23], hence showing that work-stealing with multiplicity reduces to MaxRegister.

We also consider a variation of work-stealing with multiplicity in which Take/Steal operations extracting the same task *may not be concurrent*, however, each process extracts a task *at most once*; this variant, which inserts/extracts tasks in FIFO order too, is called work-stealing with *weak multiplicity.* For this relaxation, we present an algorithm, inspired in our first solution, which uses only Read/Write instructions, is *fully fence-free* and all its operations are wait-free; furthermore each operation has constant step complexity. To our knowledge, this is the first algorithm for work-stealing having all these properties. The algorithm is obtained by reducing work-stealing with weak multiplicity to RangeMaxRegister, a relaxation of MaxRegister that we propose here, which might be of independent interest.

We show that each of our algorithms can be easily modified so that every task is extracted by a bounded number of operations, more specifically, by at most one Steal operation. In the modified algorithms, Put and Take remain the same, and a single Swap instruction is added to Steal. Also, the algorithms can be modified so that multiplicity is provided on demand, namely, a task can be taken by multiple operations only if indicated. These variants can be used in contexts where repeatable work is not allowed or needs to be restricted.

We stress that our algorithms are also wait-free solutions of relaxed versions of single-enqueue multi-dequeuer queues, namely, with multiplicity and weak multiplicity. To the best of our knowledge, together with idempotent FIFO, these are the only single-enqueue multi-dequeuer queue relaxations that have been studied. Formal specifications and correctness proofs are provided, using the *linearizability* [21] and *set-linearizable* correctness formalisms [9, 27]. Intuitively, set-linearizability is a generalization of linearizability in which several concurrent operations are allowed to be linearized at the same linearization point.

We complement our results by studying the question if there are implementations of our algorithms with good performance in practical settings. We conducted an experimental evaluation comparing our algorithms to Cilk THE, Chase-Lev and the idempotent work-stealing algorithms. In the experiments, our second algorithm exhibits a better performance than the previously mentioned algorithms, while its bounded version, with a Swap-based Steal operation, shows a lower performance but keeps competitive.

Work-stealing with multiplicity and idempotent work-stealing are closely related, but they are not the same. We observe that the idempotent work-stealing algorithms in [25] allow a task to be extracted by an unbounded number of Steal operations. This observation implies that the algorithms in [25] do not solve work-stealing with multiplicity. Therefore, the relaxations and algorithms proposed here provide stronger guarantees than idempotent work-stealing algorithms, without the need of heavy synchronization mechanisms.

Related Work. To the best of our knowledge, [25] and [26] are the only works that have been able to avoid costly synchronization mechanisms in work-stealing algorithms. The three idempotent algorithms in [25] insert/extract tasks in FIFO and LIFO orders, and as in a double-ended queue (the owner working on one side and the thieves on the other). As already mentioned, the algorithms in [26] are adaptations of the well-known Cilk THE and Chase-Lev work-stealing algorithms to the TSO model. Cilk THE and Chase-Lev (and other standard work-stealing algorithms) insert/extract tasks using double-ended queue, with the aim that the owner uses heavy synchronization mechanisms to coordinate with the thieves only when the queue is almost empty.

The experimental evaluation shows that the idempotent algorithms outperform Cilk THE and Chase-Lev. The work-stealing algorithms in [26] are fence-free adaptations of Cilk THE and Chase-Lev [12, 15] to the TSO model. The model itself guarantees that fences among Write (resp. Read) instructions are not needed, and the authors of [26] assume that write-buffers in TSO are bounded so that a fence-free coordination mechanism can be added to the Take and Steal operations of Cilk THE and Chase-Lev. The experimental evaluation in [26] shows that their algorithms outperform Cilk THE and Chase-Lev and sometimes achieve performance comparable to the idempotent work-stealing algorithms.

The notion of multiplicity was recently introduced in [10] for queues and stacks. In a queue/stack with multiplicity, an item can be dequeued/popped by several operations but only if they are concurrent. Read/Write set-linearizable algorithms for queues and stacks with multiplicity and without Read-After-Write synchronization patterns are provided in [10], and it is noted that these algorithms are also solutions for work-stealing with multiplicity; the step complexity of Enqueue/Push, which implements Put, is $\Theta(n)$ while the step complexity of Dequeue/Pop, which implements Take and Steal, is unbounded, where n denotes the number of processes. Our algorithms follow different principles than those in [10]. Our weak multiplicity relaxation of work-stealing follows the spirit of the relaxations in [10] in the sense that it requires that any algorithm for the relaxation provides "exact" responses in sequential executions, namely, the relaxation applies only if there is no contention.

As far we know, only two single-enqueue multi-dequeuer queue algorithms have been proposed [13, 22]. The algorithm in [13] is wait-free with constant step complexity and uses Read-Modify-Write instructions with consensus number 2, specifically, Swap and Fetch&Increment; however, the algorithm uses arrays of infinite length. It is explained in [13] that this assumption can be removed at the cost of increasing the step complexity to $O(n)$, where n denotes the number of processes. The algorithm in [22] is wait-free with $O(\log n)$ step complexity and uses LL/SC, a Read-Modify-Write instruction whose consensus number is ∞, and using a bounded amount of memory. Our results show that there are single-enqueue multi-dequeuer queue relaxations that can be solved using very light synchronization mechanisms.

2 Preliminaries

Model of Computation. We consider a standard concurrent shared memory system with $n \geq 2$ *asynchronous* processes, $p_0, \ldots, p_{n-1}$, which may *crash* at any time during an execution. Processes communicate with each other by invoking *atomic* instructions of *base* objects: either simple Read/Write instructions, or more powerful Read-Modify-Write instructions, such as Swap or Compare&Swap. The *index* of process p_i is i.

An *algorithm* for a high-level concurrent object T (e.g. a queue or a stack) is a distributed algorithm $\mathcal{A}$ consisting of local state machines $A_1, \ldots, A_n$. Local machine A_i specifies which instructions of base objects p_i execute in order to return a response, when it invokes a (high-level) operation of T; each of these instructions is a *step.* An *execution* of $\mathcal{A}$ is a (possibly infinite) sequence of steps, plus invocations and responses of operations of the concurrent object T; for any invocation of a process p_i, the steps of p_i between that invocation and its corresponding response (if there is one), denoted are steps that are specified by the local state machine A_i. An operation call in an execution is *complete* if both its invocation and response appear in the execution, otherwise it is *pending.*

A process is *correct* in an infinite execution if it takes infinitely many steps. An algorithm, or an operation of it, is *nonblocking* if whenever processes take steps, at least one of the invocations terminates, namely, in every infinite execution, infinitely many operations terminate [21]. An algorithm, or an operation of it, is *wait-free* if every process completes each operation in a finite number of its steps, namely, every correct process completes infinitely many operations [19]. *Bounded wait-freedom* [18] additionally requires that there is a bound on the number of steps needed to terminate. The *step complexity* of an operation is the maximum number steps a process needs to execute in order to return.

In a Read-After-Write synchronization pattern, a process first writes in a shared variable and then reads another shared variable, maybe executing other instructions in between. An algorithm, or one of its operations, is *fence-free* if it does not require any specific ordering among its steps, beyond what is implied by data dependence (e.g. the value written by a Write instruction depends on the value read by previous a Read instruction). Note that a fence-free algorithm does not use Read-After-Write synchronization patterns. In our algorithms, we use the notation $\{O_1.inst_1, \ldots, O_x.inst_x\}$ to denote that the instructions $O_1.inst_1, \ldots, O_x.inst_x$ can be executed in any order. Observe that fence instructions (also called memory barriers) are not required to correctly implement a fence-free algorithm in a concrete programming language or multicore architecture since any reordering of non-data-dependent instructions does not affect the correctness of the algorithm. For sake of simplicity in the analysis, first we will present our algorithms using base objects of infinite length and arrays of infinite length. Later we explain how we deal with this unrealistic assumptions.

Correctness Conditions. *Linearizability* [21] is the standard notion used to identify a correct implementation. Intuitively, an execution is linearizable if its (high-level) operations can be ordered sequentially, without reordering non-overlapping operations, so that their responses satisfy the specification of the implemented object. A *sequential specification* of a concurrent object T is a state machine specified through a transition function δ. Given a state q and an invocation $inv(\mathsf{op})$, $\delta(q, inv(\mathsf{op}))$ returns the tuple $(q', res(\mathsf{op}))$ (or a set of tuples if the machine is *non-deterministic*) indicating that the machine moves to state q' and the response to op is $res(\mathsf{op})$. The sequences of invocation-response tuples, $\langle inv(\mathsf{op}) : res(\mathsf{op})\rangle$, produced by the state machine are its *sequential executions.* Given an execution E, we write $\mathsf{op} <_E \mathsf{op}'$ if and only if $res(\mathsf{op})$ precedes $inv(\mathsf{op}')$ in E. Two operations are *concurrent*, denoted $\mathsf{op} \,||_E\, \mathsf{op}'$, if they are incomparable by $<_E$. The execution is *sequential* if $<_E$ is a total order.

▶ **Definition 1** (Linearizability). *Let $\mathcal{A}$ be an algorithm for a concurrent object T. A finite execution E of $\mathcal{A}$ is* linearizable *if there is a sequential execution S of T such that (1) S contains every completed operation of E and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in E, and (2) for every two completed operations* op *and* op' *in E, if* $\mathsf{op} <_E \mathsf{op}'$, *then* op *appears before* op' *in S. We say that $\mathcal{A}$ is* linearizable *if each of its finite executions is linearizable.*

Intuitively, while linearizability requires a total order on the operations, set-linearizability [9, 27] allows several operations to be linearized at the same linearization point. A *set-sequential specification* of a concurrent object differs from a sequential execution in that δ

receives as input the current state q of the machine and a set $Inv = \{inv(\mathsf{op}_1), \ldots, inv(\mathsf{op}_t)\}$ of operation invocations that happen concurrently. Thus $\delta(q, Inv)$ returns (q', Res) where q' is the next state and $Res = \{res(\mathsf{op}_1), \ldots, res(\mathsf{op}_t)\}$ are the responses to the invocations in Inv (if the machine is non-deterministic, Res is a set of sets of responses). The sets Inv and Res are called *concurrency classes.* The sequences of invocation-response concurrency classes, $\langle INV : RES\rangle$, produced by the state machine are its *set-sequential executions.* Observe that in a sequential specification all concurrency classes have a single element.

Let $\mathcal{A}$ be an algorithm for a concurrent object T. A finite execution E of $\mathcal{A}$ is *set-linearizable* if there is a set-sequential execution S of T such that (1) S contains every completed operation of E and might contain some pending operations. Inputs and outputs of invocations and responses in S agree with inputs and outputs in E, and (2) for every two completed operations op and op' in E, if $\mathsf{op} <_E \mathsf{op}'$, then op appears before op' in S. We say that $\mathcal{A}$ is *set-linearizable* if each of its finite executions is set-linearizable.

3 Work-Stealing with Multiplicity

Work-stealing with *multiplicity* is a relaxation of the usual work-stealing in which every task is extracted *at least once*, and if it is extracted by several operations, they must be *concurrent.* In the formal set-sequential specification below (and in its variant in the next section), tasks are inserted/extracted in FIFO order but it can be easily adapted to encompass other orders (e.g. LIFO). Figure 1 depicts an example of a set-sequential execution of the work-stealing with multiplicity, where concurrent Take/Steal operations can extract the same task.

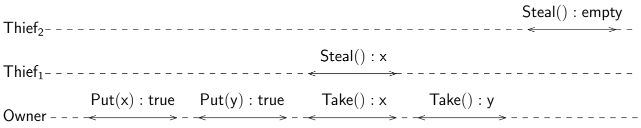


Figure 1 A set-seq. exec. of work-stealing with multiplicity.

▶ **Definition 2** ((FIFO) Work-Stealing with Multiplicity)**.** *The universe of tasks that the owner can put is* $\mathbf{N} = \{1, 2, \ldots\}$*, and the set of states* Q *is the infinite set of finite strings* $\mathbf{N}^*$*. The initial state is the empty string, denoted* ϵ*. In state* q*, the first element in* q *represents the* head *and the last one the* tail*.* $\forall q \in Q$*,* $0 \le t \le n-1$*,* $x \in \mathbf{N}$*, the transitions are:*

1. $\delta(q, \mathsf{Put}(x)) = (q \cdot x, \langle \mathsf{Put}(x) : \mathsf{true}\rangle)$.
2. $\delta(x \cdot q, \{\mathsf{Take}(), \mathsf{Steal}_1(), .., \mathsf{Steal}_t()\}) = (q, \{\langle \mathsf{Take}() : x\rangle, \langle \mathsf{Steal}_1() : x\rangle, .., \langle \mathsf{Steal}_t() : x\rangle\})$.
3. $\delta(x \cdot q, \{\mathsf{Steal}_1(), .., \mathsf{Steal}_t()\}) = (q, \{\langle \mathsf{Steal}_1() : x\rangle, .., \langle \mathsf{Steal}_t() : x\rangle\})$.
4. $\delta(\epsilon, \mathsf{Take}()) = (\epsilon, \langle \mathsf{Take}() : \mathsf{empty}\rangle)$.
5. $\delta(\epsilon, \mathsf{Steal}()) = (\epsilon, \langle \mathsf{Steal}() : \mathsf{empty}\rangle)$.

Let $\mathcal{A}$ be a linearizable algorithm for work-stealing with multiplicity. Note that items 2 and 3 in Definition 2 and the definition of set-linearizability directly imply that in every execution of $\mathcal{A}$, the number of Take/Steal operations that take the same task is at most the number of processes in the system, as the operations must be pairwise concurrent to be set-linearized together. Furthermore, every *sequential* execution of $\mathcal{A}$ looks like an “exact” solution for work-stealing, as every operation is linearized alone, by definition of set-linearizability; formally, every sequential execution of $\mathcal{A}$ is sequential execution of (FIFO) work-stealing. We call this property *sequentially-exact.* Thus, in the absence of contention, $\mathcal{A}$ provides an exact solution for work-stealing.

▶ Remark 3. Any set-lin. algorithm for work-stealing with multiplicity is sequentially-exact.

Work-Stealing with Multiplicity from MaxRegister. Here we show that work-stealing with multiplicity can be reduced to a single instance of a MaxRegister object (defined below). We will argue that our algorithm and the Read/Write wait-free MaxRegister algorithm in [3], provide a work-stealing a solution with multiplicity with logarithmic step complexity and without Read-After-Write synchronization patterns in Take and Steal.

```
Shared Variables:
    $Head$: atomic MaxRegister object initialized to 1
    $Tasks[1, 2, \ldots]$: array of atomic Read/Write objects
                 with the first two objects initialized to $\perp$
Persistent Local Variables of the Owner:
    $tail \leftarrow 0$

Operation Put($x$):
(01) $tail \leftarrow tail + 1$
(02) $\{Tasks[tail]$.Write($x$), $Task[tail + 2]$.Write($\perp$)$\}$
(03) return  true
end Put

Operation Take():
(04) $head \leftarrow Head$.MaxRead()
(05) if $head \leq tail$ then
(06)     $\{x \leftarrow Tasks[head]$.Read(), $Head$.MaxWrite($head + 1$)$\}$
(07)     return $x$
(08) return empty
end Take

Operation Steal():
(09) $head \leftarrow Head$.MaxRead()
(10) $x \leftarrow Tasks[head]$.Read()
(11) if $x \neq \perp$ then
(12)     $Head$.MaxWrite($head + 1$)
(13)     return $x$
(14) return empty
end Steal
```

Figure 2 WS-MULT: a MaxRegister-based set-lin. algorithm for work-stealing with multiplicity.

The algorithm presented in this section does not seem to have practical implications, however it will lead us to our efficient fully fence-free Read/Write work-stealing algorithm with constant step complexity in all its operations. Figure 2 presents WS-MULT, a set-linearizable algorithm for work-stealing with multiplicity. The algorithm is based on a single wait-free linearizable MaxRegister object, which provides two operations: MaxRead that returns the maximum value written so far in the object, and MaxWrite that writes a new value only if it is greater than the largest value that has been written so far. In WS-MULT, the tail of the queue is stored in the local persistent variable *tail* of the owner, while the head is stored in the shared MaxRegister $Head$. Recall that the notation in Line 2 denotes that the instructions can be executed in any order. The semantics of MaxRegister guarantees that $Head$ contains the current value of the head at all times, as a "slow" process cannot "move back" the head by writing a smaller value in $Head$. Thus, the MaxRegister $Head$ acts as a sort of barrier in the algorithm. The net effect of this is that the only way that two Take/Steal operations return the same task is because they are concurrent, reading the same value from $Head$.

Note that if only the first object in $Tasks$ is initialized to $\perp$ (and hence Put is modified accordingly), it is possible that a thief reads a value from $Tasks$ that has not been written by the owner: in an execution with a single Put(x) operation, the steps in Line 2 could be executed $Tasks[1]$.Write(x) first and then $Task[2]$.Write($\perp$) with a sequence of two Steal operations completing in between, hence the second operations reading $Tasks[2]$ which has not been written yet by the owner, which is a problem if $Tasks[2]$ contains a value distinct from non-$\perp$ value.

We also observe that the algorithm remains correct if Take is the same as Steal; Take in 2 saves a Read step when the queue is empty.

▶ **Theorem 4.** *Algorithm* WS-MULT *is a set-linearizable wait-free fence-free algorithm for work-stealing with multiplicity using atomic* Read/Write *objects and a single atomic* MaxRegister *object. All operations have constant step complexity and* Put *is fully* Read/Write*.*

When we replace $Head$ with the wait-free linearizable Read/Write MaxRegister algorithm in [3], whose step complexity is $O(\log m)$, where $m \geq 1$ is the maximum value that can be stored in the object, the step complexity of WS-MULT is bounded wait-free with logarithmic step complexity too. In the resulting algorithm at most m tasks can be inserted. Since the algorithm in [3] does not use Read-After-Write synchronization patterns (as explained in the proof of Theorem 5), the resulting algorithm does not use those patterns either.

▶ **Theorem 5.** *If $Head$ is an instance of the wait-free linearizable* Read/Write MaxRegister *algorithm in [3],* WS-MULT *is linearizable and fully* Read/Write *with* Take *and* Steal *having step complexity $O(\log m)$, where m denotes the maximum number or tasks that can be inserted in an execution.* Take *and* Steal *do not use* Read-After-Write *synchronization patterns.*

4 Work-Stealing with Weak Multiplicity

In the context of work-stealing, a logarithmic step complexity of the Take operation may be prohibitive in practice. Ideally, we would like to have constant step complexity, in all operations if possible, and using simple synchronization mechanisms. In this section, we propose work-stealing with *weak* multiplicity, which admits fully Read/Write fence-free implementations with constant step complexity in all its operations. Intuitively, weak multiplicity requires that every task is extracted at least once, but now every process extracts a task *at most once*, hence Take/Steal operations returning the same task *might not* be concurrent. Therefore, the relaxation retains the property that the number of operations that can extract the same task is at most the number of processes in the system.

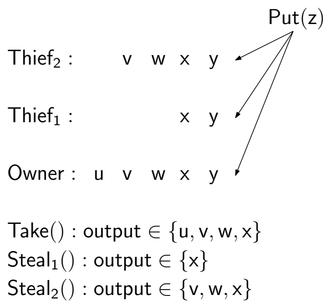

Put(z)
Thief$_2$: v w x y
Thief$_1$: x y
Owner : u v w x y
Take() : output ∈ {u, v, w, x}
Steal$_1$() : output ∈ {x}
Steal$_2$() : output ∈ {v, w, x}

Figure 3 A schematic view of weak multiplicity.

Figure 3 depicts a schematic view of a state of work-stealing with weak multiplicity. Intuitively, at any state, each process has its own *virtual* queue of tasks. When the owner inserts a new task, it concurrently places the task in all virtual queues. Therefore, in any state, for any pair of process' virtual queues, one of them is suffix of the other. A Take/Steal operation can return any task from its virtual queue that is not "beyond" the first task of the *shortest* virtual queue in the state. In the example, a Steal operation of thief p_1, denoted Steal$_1$(), can return only x, as p_1 has the shortest virtual queue in the state, while a Take operation of the owner can return any task in $\{u, v, w, x\}$, which contains any task from the beginning of its virtual queue up to x. In our algorithms, the virtual queues are implemented using a single queue.

The next *sequential* specification formally defines work-stealing with weak multiplicity. Without loss of generality, the specification assumes that p_0 is the owner, and each invocation/response of thief p_i is subscripted with its index $i \in \{1, \ldots, n-1\}$. Figure 4 shows an example of a sequential execution of work-stealing with weak multiplicity; note that Take/Steal operations are allowed to get the same item, although they are not concurrent.

▶ **Definition 6** ((FIFO) Work-Stealing with Weak Multiplicity). *The universe of tasks that the owner can put is* $\mathbf{N} = \{1, 2, \ldots\}$*, and the set of states* Q *is the infinite set of* n*-vectors of finite strings* $\mathbf{N}^* \times \ldots \times \mathbf{N}^*$*, with the property that for any two pairs of strings in a vector, one of them is a* suffix of the other. *The initial state is the vector with empty strings,* $(\epsilon, \ldots, \epsilon)$*.* $\forall (t_0, \ldots, t_{n-1}) \in Q$*, the transitions are the following:*

1. $\delta((t_0, \ldots, t_{n-1}), \mathsf{Put}(x)) = ((t_0 \cdot x, \ldots, t_{n-1} \cdot x), \langle \mathsf{Put}(x) : \mathsf{true} \rangle)$.
2. *If* $t_0 = x_1 \cdots x_j \cdot q \neq \epsilon$ *with* $j \geq 1$ *and* $x_j \cdot q$ *being the shortest string in the state (possibly with* $x_j \cdot q = \epsilon$*), then* $\delta((t_0, \ldots, t_{n-1}), \mathsf{Take}()) = \{((\widehat{t_0}, t_1, \ldots, t_{n-1}), \langle \mathsf{Take}() : x_k \rangle)\}$*, where* $k \in \{1, \ldots, j\}$ *and* $\widehat{t_0} = x_{k+1} \cdots x_j \cdot q$*.*
3. *If* $t_i = x_1 \cdots x_j \cdot q \neq \epsilon$ *with* $j \geq 1$*,* $i \in \{1, \ldots, n-1\}$ *and* $x_j \cdot q$ *being the shortest string in the state (possibly with* $x_j \cdot q = \epsilon$*), then*
   $\delta((t_0, \ldots, t_{n-1}), \mathsf{Steal}_i()) = \{((t_0, \ldots, t_{i-1}, \widehat{t_i}, t_{i+1}, \ldots, t_{n-1}), \langle \mathsf{Steal}_i() : x_k \rangle)\}$*, where* $k \in \{1, \ldots, j\}$ *and* $\widehat{t_i} = x_{k+1} \cdots x_j \cdot q$*.*
4. *If* $t_0 = \epsilon$*, then* $\delta((\epsilon, t_1, \ldots, t_{n-1}), \mathsf{Take}()) = ((\epsilon, t_1, \ldots, t_{n-1}), \langle \mathsf{Take}() : \epsilon \rangle)$*.*
5. *If* $t_i = \epsilon$ *with* $i \in \{1, \ldots, n-1\}$*, then*
   $\delta((t_0, \ldots, t_{i-1}, \epsilon, t_{i+1}, \ldots, t_{n-1}), \mathsf{Steal}_i()) = ((t_0, \ldots, t_{i-1}, \epsilon, t_{i+1}, \ldots, t_{n-1}), \langle \mathsf{Steal}_i() : \epsilon \rangle)$*.*

Observe that the second and third items in Definition 6 correspond to non-deterministic transitions in which a Take/Steal operation can extract any task of $\{x_1, \ldots, x_j\}$; the value returned can be ϵ when the shortest string in the state ($x_j \cdot q'$ in the definition) is ϵ. Furthermore, the definition guarantees that every task is extracted at least once because every Take/Steal operation can only return a task that is not "beyond" the first task in the shortest string of the state.

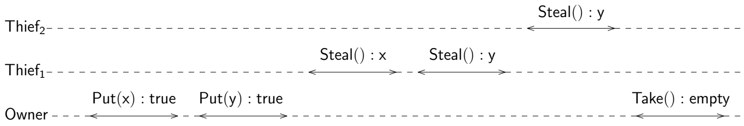


Figure 4 A seq. exec. of work-stealing with weak multiplicity.

The specification of work-stealing with concurrent multiplicity is nearly trivial, with solutions requiring almost no synchronization. A simple solution is obtained by replacing *Head* in WS-MULT with one local persistent variable *head* per process (each initialized to 1); Put remains the same and Take and Steal instead of reading from *Head*, they just locally read the current value of *head* and increment it whenever a task is taken. To avoid this kind of simple solution (which would very inefficient in practice for solving a specific problem in parallel, as essentially every process processes every task), we restrict our attention to *sequentially-exact* algorithms, namely, every sequential execution of the algorithm is a sequential execution of the specification of (FIFO) work-stealing. [2] It is easy to see that the algorithm described above does not have this property. Finally, we stress that, differently

[2] Alternatively, work-stealing with weak multiplicity can be specified using the interval-linearizability formalism in [9], which allows us to specify that a Take/Steal operation can exhibit a non-exact only in presence of concurrency. Interval-linarizability would directly imply that any interval-linearizable solution provides an exact solution in sequential executions. Roughly speaking, in interval-linearizability, operations are linearized at intervals that can overlap each other.

from work-stealing with multiplicity, two distinct non-concurrent Take/Steal can extract the same task in an execution, which can happens only if *some* operations are concurrent in the execution, due to the sequentially-exact requirement. Particularly, in our algorithms, this relaxed behaviour can occur when processes are concurrently updating the head of the queue.

Fully Read/Write Fence-Free Work-Stealing with Weak Multiplicity. We present WS-WMULT, a fully Read/Write fence-free algorithm for work-stealing with weak multiplicity. The algorithm is obtained by replacing the atomic MaxRegister object in WS-MULT, $Head$, with an atomic RangeMaxRegister object, a relaxation of MaxRegister, defined below.

We present a RangeMaxRegister algorithm that is nearly trivial, however, it allows us to solve work-stealing with weak multiplicity in an efficient manner, with implementations exhibiting good performance in practice, as we will see in Section 7. As above, to avoid trivial solutions, we focus on sequentially-exact linearizable algorithms for RangeMaxRegister, i.e. each sequential executions of the algorithm is a sequential execution of MaxRegister.

```
Shared Variables:
    $Head$: atomic Read/Write object initialized to 1
    $Tasks[1, 2, \ldots]$: array of atomic Read/Write objects
                 with the first two objects initialized to $\bot$
Persistent Local Variables of the Owner:
    $head \leftarrow 1$
    $tail \leftarrow 0$
Persistent Local Variables of a Thief:
    $head \leftarrow 1$

Operation Put($x$):
    $tail \leftarrow tail + 1$
    $\{Tasks[tail]$.Write($x$), $Tasks[tail + 2]$.Write($\bot$)$\}$
    return true
end Put

Operation Take():
    $head \leftarrow \max\{head, Head$.Read()$\}$
    if $head \leq tail$ then
        $\{x \leftarrow Tasks[head]$.Read(), $Head$.Write($head + 1$)$\}$
        $head \leftarrow head + 1$
        return $x$
    return empty
end Take

Operation Steal():
    $head \leftarrow \max\{head, Head$.Read()$\}$
    $x \leftarrow Tasks[head]$.Read()
    if $x \neq \bot$ then
        $Head$.Write($head + 1$)
        $head \leftarrow head + 1$
        return $x$
    return empty
end Steal
```

Figure 5 WS-WMULT algorithm with the RangeMaxRegister algorithm in Figure 6 inlined.

When WS-WMULT is combined with this algorithm, it becomes fully Read/Write, fence-free, linearizable, sequentially-exact and wait-free with constant step complexity. Intuitively, in RangeMaxRegister, each process has a *private* MaxRegister and whenever it invokes RMaxRead, the result lies in the range defined by the value of its private MaxRegister and the maximum among the values of the private MaxRegisters in the state. In the sequential specification of RangeMaxRegister, each invocation/response of p_i is subscripted with i.

▶ **Definition 7** (RangeMaxRegister). *The set of states Q is the infinite set of n-vectors with natural numbers, with vector $(1, \dots, 1)$ being the initial state. $\forall(r_0, \dots, r_{n-1}) \in Q$ and $i \in \{0, \dots, n-1\}$, the transitions are the following:*

1. *If $x > r_i$ then $\delta((r_0, \dots, r_{n-1}),$ $\mathsf{RMaxWrite}_i(x)) =$ $((r_0, \dots, r_{i-1}, x, r_{i+1}, \dots, r_{n-1}),$ $\langle \mathsf{RMaxWrite}_i(x) : \mathsf{true}\rangle)$, otherwise $\delta((r_0, \dots, r_{n-1}), \mathsf{RMaxWrite}(x)) =$ $((r_0, \dots, r_{n-1}), \langle \mathsf{RMaxWrite}(x) : \mathsf{true}\rangle)$.*
2. *$\delta((r_0, \dots, r_{n-1}), \mathsf{RMaxRead}_i()) = \{((r_0, \dots, r_{n-1}), \langle \mathsf{RMaxRead}_i() : x\rangle)\}$, where $x \in \{r_i, r_i + 1, \dots, \max(r_0, \dots, r_{n-1})\}$.*

As already mentioned, WS-WMULT is the algorithm obtained by replacing *Head* in WS-MULT with an atomic RangeMaxRegister object initialized to 1 (hence MaxRead and MaxWrite are replaced by RMaxRead and RMaxWrite, respectively).

▶ **Theorem 8.** WS-WMULT *is a linearizable wait-free fence-free algorithm for work-stealing with weak multiplicity using atomic* Read/Write *objects and a single atomic* RangeMaxRegister *object. All operations have constant step complexity and* Put *is fully* Read/Write.

▶ **Theorem 9.** *The algorithm in Figure 6 is a linearizable sequentially-exact wait-free fence-free algorithm for* RangeMaxRegister *using only atomic* Read/Write *objects and with constant step complexity in all its operations.*

▶ **Theorem 10.** *If Head is an instance of the algorithm in Figure 6,* WS-WMULT *is fully* Read/Write*, fence-free, wait-free, sequentially-exact and linearizable with constant step complexity in all its operations.*

Figure 5 contains an optimized version of WS-WMULT with the RangeMaxRegister algorithm in Figure 6 inlined. Since Take and Steal first RMaxRead from and then RMaxWrite to *Tail*, the algorithm remains sequentially-exact when removing Line 1 of RMaxWrite in Figure 6. Our experimental evaluation in Section 7 tested implementations of this algorithm.

```
Shared Variables:
      R: Read/Write object init. to 1
Persistent Local Var. of a Process:
      r ← 1

Operation RMaxWrite(x):
(01) r ← max{r, R.Read()}
(02) if x > r then
(03)     {r ← x, R.Write(x)}
(04) return  true
end RMaxWrite

Operation RMaxRead():
(05) r ← max{r, R.Read()}
(06) return r
end RMaxRead
```

Figure 6 A linearizable wait-free algorithm for RangeMaxRegister.

5 Bounding the Multiplicity

Here we discuss simple variants of our algorithms that bound the number of operations that can extract the same task. We only discuss the case of WS-MULT as the variants for WS-WMULT are similar.

Bounding multiplicity. The modification consists in having an extra array A of the same length of $Tasks$, with its first two entries initialized to true. Steal is modified as follows: after Line 11, a thief performs $A[head]$.Swap(false), and it executes Lines 12 and 13 only if the Swap successfully takes the true value in $A[head]$; otherwise, it goes to Line 9 to start over. The modified algorithm guarantee that no two distinct Steal operations take the same task, however, a Take and a Steal can take the same task. Note that Steal is only nonblocking in the modified algorithm. This *bounded* variant of WS-MULT is denoted B-WS-MULT. The new algorithm is a set-linearizable solution to the variant of work-stealing with multiplicity, Definition 2, where every concurrency class has at most one Take and one Steal that return the same task.

Removing multiplicity. The Take operation of B-WS-MULT can be modified similarly to obtain an algorithm for exact (FIFO) work-stealing, i.e., every task is taken *exactly* once (Definition 2 with singleton concurrency classes). The modified Take operation remains wait-free.

Multiplicity on demand. Consider a variant of Definition 2 in which a task x encodes if it can be executed by several processes, denoted mult(x), or it has to be executed by a single process, denoted ¬mult(x) (in practice this can be done, for example, by stealing a bit from the task representation). Then, WS-MULT can be modified to have multiplicity *on demand*. In the modified Take operation, after executing the operations in Line 6, the owner tests if mult(x) holds, and if so, it returns x; otherwise, it performs $Tasks[head]$.Swap($\top$), and then returns x only if the Swap successfully takes the task in $Tasks[head]$, else it goes to Line 4 to start over. In the modified Take operation, after Line 10, a thief checks if $x = \bot$, and if so, it returns empty. Then, it checks if $x \neq \top$ and mult(x) holds, and if so it returns x. Otherwise, $x \neq \top$ and ¬mult(x) holds, and the thief performs $Tasks[head]$.Swap($\top$) and returns x only if the Swap returns a value distinct from $\top$, else it goes to Line 9 to start over. In the resulting algorithm, if mult(x) holds, x is taken by one operation. The modified Take operations remains wait-free but the modified Steal operation is only nonblocking.

6 Coping with realistic assumptions

We have presented our algorithms assuming all base objects used for manipulating/implementing the head and the tail are able to store values of unbounded length. However, we can assume that these base objects can store only 64-bit values. This makes our algorithms *bounded* as at most 2^{64} tasks can be inserted, but arguably this number is enough in any real application. We also have assumed that there is an array of infinite length where tasks are stored. We now discuss two approaches to remove this assumption; both approaches have been used in previous algorithms (e.g. [1, 2, 25, 16, 30]). We only discuss the case of WS-MULT as the other cases are handled in the same way.

In the first approach, the algorithm starts with $Tasks$ pointing to an array of finite fixed length, with its two first objects initialized to $\bot$; each time the owner detects the array is full (i.e. when $tail$ is larger the length $Tasks$), in the middle of a Put operation, it creates a new array A, duplicating the previous length, copies the previous content to A, initializes the next two objects to $\bot$, points $Tasks$ to A and finally continues executing the algorithm. Although the modified Put operation remains wait-free, its step complexity is unbounded.

In the second approach, $Tasks$ is implemented with a linked list with each node having a fixed length array. Initially, $Tasks$ consists of a single node, with the first two objects of its array initialized to $\bot$. When the owner detects that all entries in the linked list have been used, in the middle of a Put operation, it creates a new node, initializes the first two objects to ⊥, links the new node to the end of the list and continues executing the algorithm. An index of $Tasks$ is now made of a tuple: a pointer to a node of the linked list and an node-index array. Thus, any pair of nodes can be easy compared (first pointer nodes, then node-indexes) and incrementing an index can be easily performed too (if the node-index is the last one, the pointer moves forward and the node-index is set to one, otherwise only the node-index is incremented). The modified Put operation remains wait-free with constant step complexity. Our experimental evaluation in Section 7 shows the second approach performs better than the first one when solving a problem of concurrent nature.

7 Experiments

Here we discuss the outcome of the experiments we have conducted to evaluate the performance of WS-WMULT and its bounded version, B-WS-WMULT (see Section 5). Based on the approaches discussed in Section 6, we implemented two versions of the algorithms, one using arrays and another one using linked lists. We mainly discuss the results of the version based on linked lists, since it exhibited better performance than the version based on arrays. WS-WMULT and B-WS-WMULT were compared to the following algorithms: Cilk THE [12], Chase-Lev [15], and the three Idempotent Work-Stealing algorithms [25].

Platform and Environment. The experiments were executed in two machines with distinct characteristics. The first machine has an Intel Kaby Lake processor (i7–7700HQ, with four cores where each core has two threads of execution) and 16GB RAM. This machine was dedicated only to the experiments. The second one is a machine with four Intel Xeon processors (E7–8870 v3 processor) and 3TB RAM; each processor has 18 cores, with each core executing two threads, so in total it has 72 cores where 144 threads can be executed in parallel. This machine was shared with other users, where the process executions are closer to a usual situation, namely, resources (CPU and memory) are shared by all users. We will focus mainly on the results of the experiments in the Core i7 processor, and additionally will use the results of the experiments in Intel Xeon to complement our analysis. All algorithms were implemented in the Java platform (OpenJDK 1.8.0_275) in order to test them in a cross-platform computing environment.

Methodology. To analyze the performance of the algorithms, we divided the analysis into the next two benchmarks, which have been used in [15, 25, 26]: (1) *Zero cost experiments.* (2) *Irregular graph application.* Below, we explain in detail how we implement each benchmark.

Zero cost experiments. We measure the time required for performing a sequence of operations provided by the work-stealing algorithms. We measure the time needed for Put-Take operations, where the owner performs a sequence of Put operations followed by an equal number of Takes. Differently from [25, 26], we also measure the time for Put-Steal operations. In both experiments, The number of Put operations is $10,000,000$, followed by the same number of Take or Steal operations; no operation performs any work associated to a task. The idea of the zero-cost experiments is to show that the presence of heavy synchronization mechanisms of an algorithm slows down the computation even in sequential executions.

Irregular graph application. We consider the spanning-tree problem to evaluate the performance of each algorithm. It is used to measure the speed-up of the computation by the parallel exploration of the input graph. This problem is used in [25, 26] to evaluate their work-stealing algorithms. We refer the reader to [6] for a detailed description of the algorithm.

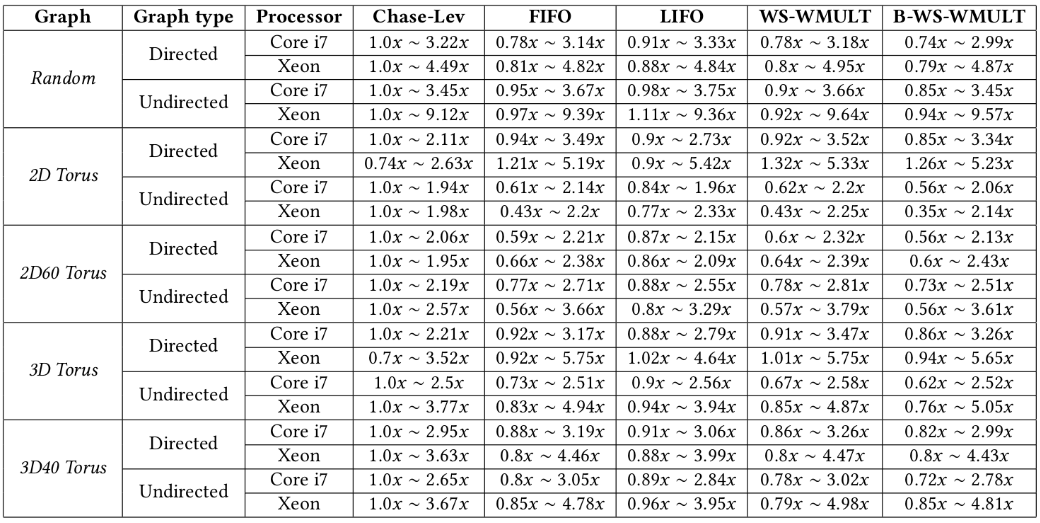
The spanning tree algorithm already uses a form of work-stealing to ensure load-balancing, and we adapted it to work with all tested work-stealing implementations. We implemented all algorithms in [15, 25, 26] in Java as the experiments in those paper were perfomed in C/C++.

Further, the algorithms were tested on many types of directed and undirected graphs. We use the following graphs: *2D Torus*, *2D60 Torus*, *3D Torus*, *3D40 Torus* and *Random*. All graphs are represented using the adjacency lists representation. Each experiment consists of the following: take one of the previous graphs, specify its parameters and run the spanning-tree algorithm with one of the work-stealing algorithms (Cilk THE, Chase-Lev, Idempotent FIFO, Idempotent LIFO, Idempotent DEQUE, WS-WMULT or B-WS-WMULT); the spanning tree algorithm is executed five times with each work-stealing algorithm, using the same root and the same graph. In each execution, we test the performance obtained by increasing the number of threads from one to the total supported by the processor and registering the duration of every execution. For each work-stealing algorithm and number of threads, the fastest and slowest executions are discarded (similarly to [25]), and the average of the remaining three is calculated. All results are normalized with respect to Chase-Lev with a single thread, as in [25, 26]. In the Appendix, we discuss about the initial length of arrays and lengths of node-arrays in the experiments, as well as the number of vertices used in the graphs.

Results. We present a summary of the results before explaining them in detail: (1) *Zero cost experiments.* WS-WMULT has a similar performance than other algorithms in the Put-Take experiment. However, WS-WMULT exhibited better performance in Put-Steal experiment. This happens because the Steal operation of any other algorithm uses costly primitives, like Compare&Swap or Swap. B-WS-WMULT has the worst performance in the Put-Steal experiment, due to the management of the additional array for marking a task as taken. However, interestingly this result does not preclude the algorithm for exhibiting a competitive performance in the next benchmark. (2) *Irregular graph application.* In general WS-WMULT has better performance than any other algorithm and in particular it performed better than Idempotent FIFO in virtually all cases. B-WS-WMULT has a lower performance than WS-WMULT but still competitive respect to Idempotent algorithms. Below we discuss in detail the results of the zero cost experiments and the irregular graph application, in both cases omitting the results of Cilk THE because its performance was similar to that of Chase-Lev. Similarly, we omit the results of Idempotent DEQUE since in general it had the worst performance among the Idempotent algorithms.

Zero cost experiments Figure 9a depicts the result of the Put-Take experiment in the Intel Core i7 processor. The results of Cilk THE and the idempotent algorithms are similar than those presented [25]. As for WS-WMULT, the time required for Put operations was similar than that of Idempotent LIFO, and slightly faster than Idempotent FIFO. Considering the whole experiment, WS-WMULT was faster than any other algorithm. The results show a significant speed-up, where the gain is between 6% and 22% respect to the other algorihtms. For the case of B-WS-WMULT, it required about twice the time of any other algorithm. This poor performance is due to the use of an extra boolean array for bounding multiplicity. Particularly, the Put operation writes three entries of an array (two entries of $Tasks$ and one of the extra boolean array), differently from WS-WMULT's Put operations that writes only two (both of $Tasks$). The results of the Put-Steal experiment in the processor Intel Core i7 are shown in Figure 9b, where, as expected, Put operations exhibited a similar performance as in the Put-Take experiment. In the case of Steal operations, we observe that Chase-Lev, Idempotent FIFO, and WS-WMULT are faster than Idempotent LIFO and B-WS-WMULT. In particular, WS-WMULT is the fastest among all algorithms, which is expected as it does not use fences and Read-Modify-Write instructions. We observe a gain between 11% and 37% compared to other algorithms. As for total time, the speed-up was between 19% and 40%. For B-WS-WMULT, we have a similar result than the one in the Puts-Takes experiment. Finally, the array-based implementation of WS-WMULT, denoted WS_WMULT_ARRAY in the figures, performed much better than any other algorithm, with a speed-up between 49% and 66% respect to the other algorithms in the Put-Steal experiment and for the Put-Take experiment, the array-based implementation is better by 42% - 52%. This implementation performs better than the linked-list version of WS-WMULT because the latter creates a large number of arrays during the execution. In the next experiment, we do not discuss in detail the result of the array-based implementations of our algorithms as the linked-list implementations have a similar performance, and sometimes outperforming them.

Irregular graph application. Table 7 displays the minimum and maximum speed-up achieved by each algorithm, normalized with respect to the one-thread execution of Chase-Lev, for every graph and processor. Overall, WS-WMULT had a better performance with maximum speed-up of 9.64 (in random undirected graph), corresponding to a cap gain of 5% respect to the maximum speed-up achieved by the other algorithms in the Xeon processor. With respect to the other algorithms, we observe an improvement range between 2% and 20%. B-WS-WMULT always shown a similar performance than Idempotent FIFO algorithm; we expected this behavior as both use expensive Read-Modify-Write instructions in Steal. Below we discuss the results only for the 2D Torus, in the appendix we discuss all other graphs.



Graph	Graph type	Processor	Chase-Lev	FIFO	LIFO	WS-WMULT	B-WS-WMULT
Random	Directed	Core i7	$1.0x \sim 3.22x$	$0.78x \sim 3.14x$	$0.91x \sim 3.33x$	$0.78x \sim 3.18x$	$0.74x \sim 2.99x$
		Xeon	$1.0x \sim 4.49x$	$0.81x \sim 4.82x$	$0.88x \sim 4.84x$	$0.8x \sim 4.95x$	$0.79x \sim 4.87x$
	Undirected	Core i7	$1.0x \sim 3.45x$	$0.95x \sim 3.67x$	$0.98x \sim 3.75x$	$0.9x \sim 3.66x$	$0.85x \sim 3.45x$
		Xeon	$1.0x \sim 9.12x$	$0.97x \sim 9.39x$	$1.11x \sim 9.36x$	$0.92x \sim 9.64x$	$0.94x \sim 9.57x$
2D Torus	Directed	Core i7	$1.0x \sim 2.11x$	$0.94x \sim 3.49x$	$0.9x \sim 2.73x$	$0.92x \sim 3.52x$	$0.85x \sim 3.34x$
		Xeon	$0.74x \sim 2.63x$	$1.21x \sim 5.19x$	$0.9x \sim 5.42x$	$1.32x \sim 5.33x$	$1.26x \sim 5.23x$
	Undirected	Core i7	$1.0x \sim 1.94x$	$0.61x \sim 2.14x$	$0.84x \sim 1.96x$	$0.62x \sim 2.2x$	$0.56x \sim 2.06x$
		Xeon	$1.0x \sim 1.98x$	$0.43x \sim 2.2x$	$0.77x \sim 2.33x$	$0.43x \sim 2.25x$	$0.35x \sim 2.14x$
2D60 Torus	Directed	Core i7	$1.0x \sim 2.06x$	$0.59x \sim 2.21x$	$0.87x \sim 2.15x$	$0.6x \sim 2.32x$	$0.56x \sim 2.13x$
		Xeon	$1.0x \sim 1.95x$	$0.66x \sim 2.38x$	$0.86x \sim 2.09x$	$0.64x \sim 2.39x$	$0.6x \sim 2.43x$
	Undirected	Core i7	$1.0x \sim 2.19x$	$0.77x \sim 2.71x$	$0.88x \sim 2.55x$	$0.78x \sim 2.81x$	$0.73x \sim 2.51x$
		Xeon	$1.0x \sim 2.57x$	$0.56x \sim 3.66x$	$0.8x \sim 3.29x$	$0.57x \sim 3.79x$	$0.56x \sim 3.61x$
3D Torus	Directed	Core i7	$1.0x \sim 2.21x$	$0.92x \sim 3.17x$	$0.88x \sim 2.79x$	$0.91x \sim 3.47x$	$0.86x \sim 3.26x$
		Xeon	$0.7x \sim 3.52x$	$0.92x \sim 5.75x$	$1.02x \sim 4.64x$	$1.01x \sim 5.75x$	$0.94x \sim 5.65x$
	Undirected	Core i7	$1.0x \sim 2.5x$	$0.73x \sim 2.51x$	$0.9x \sim 2.56x$	$0.67x \sim 2.58x$	$0.62x \sim 2.52x$
		Xeon	$1.0x \sim 3.77x$	$0.83x \sim 4.94x$	$0.94x \sim 3.94x$	$0.85x \sim 4.87x$	$0.76x \sim 5.05x$
3D40 Torus	Directed	Core i7	$1.0x \sim 2.95x$	$0.88x \sim 3.19x$	$0.91x \sim 3.06x$	$0.86x \sim 3.26x$	$0.82x \sim 2.99x$
		Xeon	$1.0x \sim 3.63x$	$0.8x \sim 4.46x$	$0.88x \sim 3.99x$	$0.8x \sim 4.47x$	$0.8x \sim 4.43x$
	Undirected	Core i7	$1.0x \sim 2.65x$	$0.8x \sim 3.05x$	$0.89x \sim 2.84x$	$0.78x \sim 3.02x$	$0.72x \sim 2.78x$
		Xeon	$1.0x \sim 3.67x$	$0.85x \sim 4.78x$	$0.96x \sim 3.95x$	$0.79x \sim 4.98x$	$0.85x \sim 4.81x$

Figure 7 Summary of speedups of each algorithm. Only maximum and minimum are shown.

2D Torus. In the case of 2D Torus (and similarly for the 2D60 Torus), we observe a pattern: our algorithms showed a bad performance with one or two threads, but when the number of threads increases, our algorithms improved the speed up of Idempotent LIFO and Chase-Lev. Additionally, the performance showed by Idempotent FIFO is between the registered by WS-WMULT and B-WS-WMULT. We can see this behavior in the Figure 10a. The results show the following: for the 2D Directed Torus, the gains range of WS-WMULT is from 1% to 40% with respect other algorithms. In a similar comparison, for the 2D Undirected Torus, the gains range of WS-WMULT is of 3% to 12% and for the 2D60 Undirected Torus and its Directed version, the gains are between 5% to 11% and 4% to 22% respectively. This behavior is consistent in both machines. In most of the of executions, WS-WMULT has a better performance than other algorithms (see Figure 10b).

Final remark. In our parallel spanning-tree experiments, only a small fraction of the tasks are repeated. One reason for this to happen is that methods of WS-WMULT are short and simple, and hence very often a process has enough time in a core to grab a task only for itself (a task can be taken several times olnly in specific concurrent scenarios). In other words, since the algorithm is very simple, an optimistic approach gives exclusive access to a task, most of the time.

8 Final Discussion

We have studied two relaxations for work-stealing, called multiplicity and weak multiplicity. Both of them allow a task to be extracted by more than one Take/Steal operation but each process can take the same task at most once; however, the relaxation can arise only in presence of concurrency. We presented fully Read/Write wait-free algorithms for the relaxations. All algorithms are devoid of Read-After-Write synchronization patterns and the algorithm for the weak multiplicity case is also fully fence-free with constant step complexity. To our knowledge, this is the first time work-stealing algorithms with these properties have been proposed, evading the impossibility result in [4] in all operations. From the theoretical perspective of the consensus number hierarchy, we have shown that work-stealing with multiplicity and weak multiplicity lays at the lowest level of the hierarchy [19]. We also argued that the idempotent work-stealing [25] do not solve work-stealing with multiplicity (see Section 7 in the Appendix), therefore the relaxations and algorithms proposed here provide stronger guarantees. We also have performed an experimental evaluation comparing our work-stealing solutions to Cilk THE, Chase-Lev and idempotent work-stealing algorithms. The experiments show that our WS-WMULT algorithm exhibits a better performance than the previously mentioned algorithms, while its bounded version, B-WS-WMULT, with a Swap-based Steal operation, shows a lower performance than WS-WMULT, but keeps a competitive performance with respect to the other algorithms. All our results together show that one of the simplest synchronization mechanisms suffices to solve non-trivial coordination problems, particularly, a relaxation of work-stealing that helps with computing a spanning tree.

For future research, we are interested in developing algorithms for work-stealing with multiplicity and weak multiplicity that insert/extract tasks in orders different from FIFO. Also, it is interesting to explore if the techniques in our algorithms can be applied to efficiently solve relaxed versions of other concurrent objects. For example, it is worthwhile to explore if a multi-enqueuer multi-dequeuer queue with multiplicity (resp. weak multiplicity) can be obtained by manipulating the tail with a MaxRegister (resp. RangeMaxRegister) object, like the head is manipulated in our algorithms.

References

1 Dolev Adas and Roy Friedman. Brief announcement: Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 50:1–50:3, 2020. doi:10.4230/LIPIcs.DISC.2020.50.

2 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, pages 395–410, 2010. doi:10.1007/978-3-642-17653-1_29.

3 James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012. doi:10.1145/2108242.2108244.

4 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 487–498, 2011. doi:10.1145/1926385.1926442.

5 Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009. doi:10.1109/TPDS.2008.105.

6 David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004. doi:10.1109/IPDPS.2004.1302951.

7 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995*, pages 207–216, 1995. doi:10.1145/209936.209958.

8 Armando Castañeda and Miguel Piña. Fully read/write fence-free work-stealing with multiplicity. *CoRR*, abs/2008.04424, 2020. arXiv:2008.04424.

9 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.

10 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Relaxed queues and stacks from read/write operations. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, pages 13:1–13:19, 2020. doi:10.4230/LIPIcs.OPODIS.2020.13.

11 Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538, 2005. doi:10.1145/1094811.1094852.

12 David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 21–28, 2005. doi:10.1145/1073970.1073974.

13 Matei David. A single-enqueuer wait-free queue implementation. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 132–143, 2004. doi:10.1007/978-3-540-30186-8_10.

14 Christine H. Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, 2001. URL: http://www.usenix.org/publications/library/proceedings/jvm01/full_papers/flood/flood.pdf.

15 Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 212–223, 1998. doi:10.1145/277650.277725.

16 Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Comput.*, 18(3):189–207, 2006. doi:10.1007/s00446-005-0144-5.

17 Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 280–289, 2002. doi:10.1145/571825.571876.

18 Maurice Herlihy. Impossibility results for asynchronous PRAM (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, Hilton Head, South Carolina, USA, July 21-24, 1991*, pages 327–336, 1991. doi:10.1145/113379.113409.

19 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

20 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

21 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

22 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th International Conference, Hyderabad, India, December 15-18, 2005, Proceedings*, pages 408–419, 2005. doi:10.1007/11590156_33.

23 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000. doi:10.1137/S0097539797317299.

24 Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*, pages 36–43, 2000. doi:10.1145/337449.337465.

25 Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 45–54, 2009. doi:10.1145/1504176.1504186.

26 Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded TSO processors. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 413–426, 2014. doi:10.1145/2541940.2541987.

27 Gil Neiger. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, page 396, 1994. doi:10.1145/197917.198176.

28 Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 13–24, 2007. doi:10.1109/HPCA.2007.346181.

29 Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010. doi:10.1145/1785414.1785443.

30 Chaoran Yang and John M. Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, pages 16:1–16:13, 2016. doi:10.1145/2851141.2851168.

A Idempotent $\neq$ Multiplicity

Idempotent work-stealing is (only) informally defined in [25] as: every task is extracted *at least once*, instead of *exactly once* (in some order). Here we explain that these algorithms does not implement work-stealing with multiplicity, neither its non-concurrent variant. While in our relaxations every process extracts a task at most once, and hence the number of distinct operations that extract the same task is at most the number of processes in the system, in idempotent work-stealing a task can be extracted an unbounded number of times, and moreover, a thief can extract the same task an unbounded number of times.

```
   Structures:
     Task: task information
     TaskArrayWithSize:
       size: integer
       array: array of Task
     FifoIwsq:
       head: integer;
       tail: integer;
       tasks: TaskArrayWithSize

   constructor FifoIwsq(integer size) {
     head := 0;
     tail := 0;
     tasks := new TaskArrayWithSize(size);
   }

   void put(Task task) {
     Order write at 4 before write at 5
1:   h := head;
2:   t := tail;
3:   if (t = h+tasks.size) {expand(); goto 1;}
4:   tasks.array[t%tasks.size] := task;
5:   tail := t+1;
   }

   TaskInfo take() {
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   task := tasks.array[h%tasks.size];
5:   head := h+1;
6:   return task;
   }

   TaskInfo steal() {
     Order read in 1 before read in 2
     Order read in 1 before read in 4
     Order read in 5 before CAS in 6
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   a := tasks;
5:   task := a.array[h%a.size];
6:   if !CAS(head,h,h+1) goto 1;
7:   return task;
   }
```

Figure 8 Idempotent FIFO work-stealing [25].

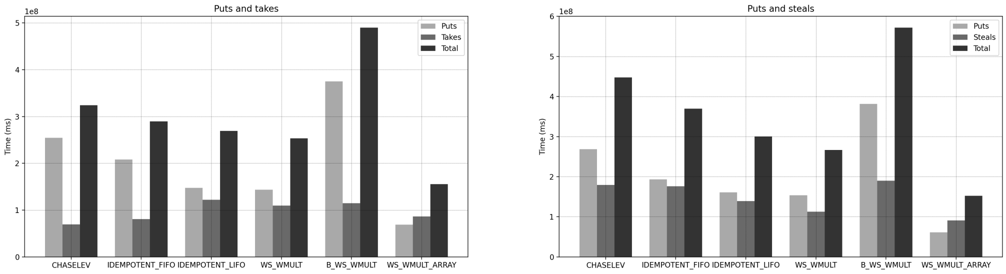
Figure 8 depicts the FIFO idempotent work-stealing algorithm in [25]. For every integer $z > 0$, we describe an execution of the algorithm in which, for every $k \in \{1, \dots, z\}$, there is a task that is extracted by $\Theta(k)$ distinct operations (possibly by the same thief), with only one of them being concurrent with the others.

1. Let the owner execute alone z times Put. Thus, there are z distinct tasks in $tasks$
2. Let $r = z$.
3. The owner executes Take and stops before executing Line 5, i.e. it is about to increment $head$.
4. In some order, the thieves sequentially execute r Steal operations; note these Steal operations return the r tasks in $tasks[0, \dots, r-1]$.
5. We now let the owner increment $head$. If $r > 1$, go to step 3 with r decremented by one, else, end the execution.

Observe that in the execution just described, the task in $tasks[i]$, $i \in \{0, \dots, z-1\}$, is extracted by a Take operation and by $i+1$ distinct non-concurrent Steal operations (possible by the same thief). Thus, the task is extracted $\Theta(i)$ distinct times. Since z is any positive integer, we conclude that there is no bound on the number of times a task can be extracted.

A similar argument works for the other two idempotent work-stealing algorithms in [25]. In the end, this happens in all algorithms because tasks are not marked as taken in the shared array where they are stored. Thus, when the owner takes a task and experience a delay before updating the head/tail, all concurrent modifications of the head/tail performed by the thieves are overwritten once the owner completes its operation, hence leaving all taken tasks ready to be taken again.

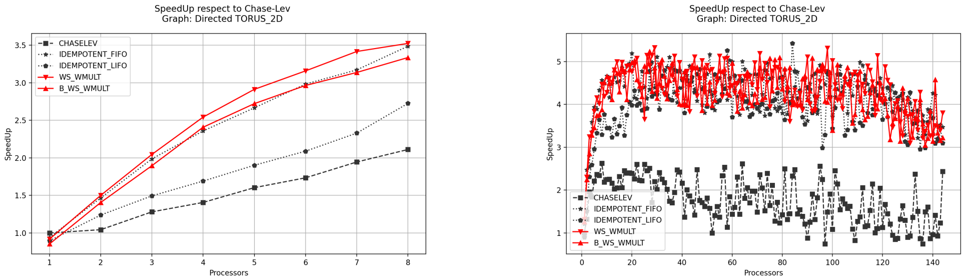
B Additional Figures



(a) Results of experiment for puts and takes.

(b) Results of experiment for puts and steals.

Figure 9 Zero cost experiments, less time is better.



(a) Speed up on Intel Core i7.

(b) Speed up on Intel Xeon.

Figure 10 Speed-ups of algorithms using directed torus 2D graph.

Optimal Error-Free Multi-Valued Byzantine Agreement

Jinyuan Chen ✉
Louisiana Tech University, Ruston, LA, USA

Abstract

Byzantine agreement (BA) is a distributed consensus problem where n processors want to reach agreement on an ℓ-bit message or value, but up to t of the processors are dishonest or faulty. The challenge of this BA problem lies in achieving agreement despite the presence of dishonest processors who may arbitrarily deviate from the designed protocol. In this work by using coding theory, together with graph theory and linear algebra, we design a **co**ded BA protoc**ol** (termed as COOL) that achieves consensus on an ℓ-bit message with optimal *resilience*, asymptotically optimal *round complexity*, and asymptotically optimal *communication complexity* when $\ell \geq t \log t$, simultaneously. The proposed COOL is a *deterministic* BA protocol that is guaranteed to be correct in all executions (*error free*) and does not rely on cryptographic technique such as signatures, hashing, authentication and secret sharing (*signature free*). It is secure against computationally unbounded adversary who takes full control over the dishonest processors (*information-theoretic secure*). The main idea of the proposed COOL is to use a carefully-crafted error correction code that provides an efficient way of exchanging "compressed" information among distributed nodes, while keeping the ability of detecting errors, masking errors, and making a consistent and validated agreement at honest distributed nodes. We show that our results can also be extended to the setting of Byzantine broadcast, aka Byzantine generals problem, where the honest processors want to agree on the message sent by a leader who is potentially dishonest. The results reveal that *coding* is an effective approach for achieving the fundamental limits of Byzantine agreement and its variants. Our protocol analysis borrows tools from coding theory, graph theory and linear algebra.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine agreement, information-theoretic security, error correction codes

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.17

Related Version *Full Version*: `https://arxiv.org/pdf/2009.10965.pdf`

Funding *Jinyuan Chen*: This work was supported in part by the NSF under Grant OIA-1946231 and in part by the Louisiana Board of Regents under Grant 32-4121-40336.

1 Introduction

Byzantine agreement (BA), as originally proposed by Pease, Shostak and Lamport in 1980, is a distributed consensus problem where n processors want to reach agreement on some message (or value), but up to t of the processors are dishonest or faulty [52]. The challenge of this BA problem lies in achieving agreement despite the presence of dishonest processors who may arbitrarily deviate from the designed protocol. One variant of the problem is Byzantine broadcast (BB), aka Byzantine generals problem, where the honest processors want to agree on the message sent by a leader who is potentially dishonest [39]. Byzantine agreement and its variants are considered to be the fundamental building blocks for distributed systems and cryptography including Byzantine-fault-tolerant (BFT) distributed computing, distributed storage, blockchain protocols, state machine replication and voting, just to name a few [52, 39, 26, 45, 23, 41, 25, 42, 48, 50, 9, 43, 1, 13, 57, 49, 53, 27, 58].

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 17; pp. 17:1–17:19
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Comparison of the proposed and some other error-free synchronous BA protocols.

Protocols	Resilience	Communication	Round	Error Free	Signature Free
ℓ-1-bit	$n \geq 3t+1$	$\Omega(n^2\ell)$	$\Omega(\ell t)$	yes	yes
[41]	$n \geq 3t+1$	$O(n\ell + n^4\sqrt{\ell} + n^6)$	$\Omega(\sqrt{\ell} + n^2)$	yes	yes
[25]	$n \geq 3t+1$	$O(n\ell + n^4)$	$O(t)$	yes	yes
[42]	$n \geq 3t+1$	$O(n\ell + n^4)$	$O(t)$	yes	yes
[48]	$n \geq 3t+1$	$O(n\ell + n^3)$	$O(t)$	yes	yes
Proposed	$n \geq 3t+1$	$O(\max\{n\ell, nt\log t\})$	$O(t)$	yes	yes

To solve the Byzantine agreement problem, a designed protocol needs to satisfy the following conditions: every honest processor eventually outputs a message and terminates (*termination*); all honest processors output the same message (*consistency*); and if all honest processors hold the same initial message then they output this initial message (*validity*). A protocol that satisfies the above three conditions in all executions is said to be *error free*. The quality of a BA protocol is measured primarily by using three parameters:

- Resilience: the number of processors n as a function of t allowed.
- Round complexity: the number of rounds of exchanging information, denoted by r.
- Communication complexity: the total number of communication bits, denoted by b.

For any error-free BA protocol, the known lower bounds on those parameters are respectively

$$n \geq 3t+1 \quad \text{(cf. [52, 39])}, \quad r \geq t+1 \quad \text{(cf. [22, 20])}, \quad b \geq \Omega(\max\{n\ell, nt\}) \ \text{(cf. [19, 23])}$$

where ℓ denotes the length of message. It is worth mentioning that, in practice the consensus is often required for *multi-valued* message rather than just single-bit message [23, 41, 25, 42, 48, 50, 51]. For example, in BFT consensus protocols of Libra (or Diem) and Hyperledger Fabric proposed by Facebook and IBM respectively, the message being agreed upon could be a transaction or transaction block with size scaled from $1KB$ to $1MB$ [4, 59, 28, 33, 60, 5, 3]. Also, in practice the consensus is often expected to be 100% secure and error free in mission-critical applications such as online banking and smart contracts [8, 30, 44, 12, 18].

The multi-valued BA problem of achieving consensus on an ℓ-bit message could be solved by invoking ℓ instances of 1-bit consensus in sequence, which is termed as ℓ-1-bit scheme. However, this scheme will result in communication complexity of $\Omega(n^2\ell)$ bits, because $\Omega(n^2)$ is the lower bound on communication complexity of 1-bit consensus given $n \geq 3t+1$ [16, 7, 19]. In 2006, Fitzi and Hirt provided a *probabilistically correct* multi-valued BA protocol by using a hashing technique, which results in communication complexity of $O(n\ell + n^3(n+\kappa))$ bits for some constant κ [23]. In 2011, Liang and Vaidya provided an *error-free* BA protocol with communication complexity $O(n\ell + n^4\sqrt{\ell} + n^6)$ bits, which is optimal when $\ell \geq n^6$ [41]. However, in the regime of $\ell < n^6$, this communication complexity is sub-optimal. The result of [41] was improved recently in [25], [42] and [48]. Specifically, the communication complexities of the BA protocols proposed in [25], [42] and [48] are $O(n\ell+n^4)$ bits, $O(n\ell + n^4)$ bits, and $O(n\ell + n^3)$ bits, respectively. Although the communication complexity has been improved in [25], [42] and [48], the achievable performance is still sub-optimal in the regime of $\ell < n^2$. In some previous work, randomized algorithms were proposed to reduce communication and round complexities but the termination cannot be 100% guaranteed [21, 50, 2, 11, 15, 35, 46, 47]. In some other work, the protocols were designed with cryptographic technique such as signatures, hashing, authentication and secret sharing [54, 35, 1, 20]. However, the protocols with such cryptographic technique are vulnerable to attacks from the adversary with very high computation power, e.g., using

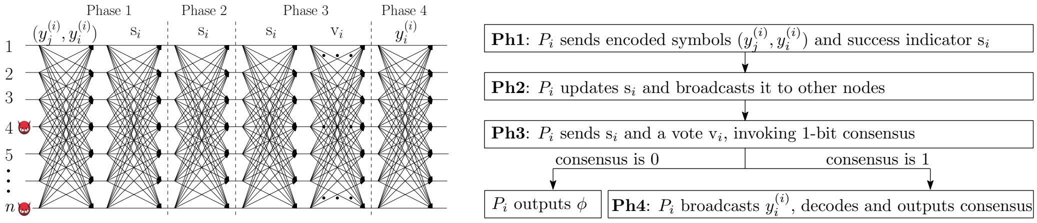

Phase 1
Phase 2
Phase 3
Phase 4
$(y_j^{(i)}, y_i^{(i)})$
s_i
s_i
s_i
v_i
$y_i^{(i)}$
1
2
3
4
5
n
Ph1: P_i sends encoded symbols $(y_j^{(i)}, y_i^{(i)})$ and success indicator s_i
Ph2: P_i updates s_i and broadcasts it to other nodes
Ph3: P_i sends s_i and a vote v_i, invoking 1-bit consensus
consensus is 0
consensus is 1
P_i outputs ϕ
Ph4: P_i broadcasts $y_i^{(i)}$, decodes and outputs consensus

Figure 1 Four-phase operation and block diagram at Processor i (P_i) of the proposed COOL.

supercomputer or quantum computer possibly available in the future, and hence not error free. A protocol that doesn't rely on cryptographic technique mentioned above is said to be *signature free*. A protocol is said to be *information-theoretic secure* if it is secure against computationally unbounded adversary who takes full control over the dishonest processors.

In this work we focus on the fundamental limits of error-free multi-valued Byzantine agreement. Specifically by using coding theory, together with graph theory and linear algebra, we are able to design an error-free signature-free information-theoretic-secure multi-valued BA protocol (named as COOL) with optimal *resilience*, asymptotically optimal *round complexity*, and asymptotically optimal *communication complexity* when $\ell \geq t \log t$, simultaneously (see Table 1), focusing on the BA setting with synchronous communication network. In a nutshell, carefully-crafted error correction codes provide an efficient way of exchanging "compressed" information among distributed processors, while keeping the ability of detecting errors, masking errors, and making a consistent agreement at honest distributed processors.

The main difference between the protocols of [41, 25, 42, 48] and our proposed protocol is that, while coding is also used, in the protocols of [41, 25, 42, 48] each distributed processor needs to send an n-bit information to all other processors after generating a graph based on the exchanged information, which results in a communication complexity for this step of at least $\Omega(n^3)$ bits. This is a limitation in the protocols of [41, 25, 42, 48]. Our proposed protocol, which is carefully designed by using coding theory, graph theory and linear algebra, avoids the limitation appeared in [41, 25, 42, 48]. Specifically, our proposed protocol consists of at most four phases (see Fig. 1 and Section 4). After the third phase, it is guaranteed that at most one group of honest processors output the same non-empty value and the size of this group is bigger than the group size of dishonest processors. In this way honest processors can calibrate their values based on majority rule and error-correction decoding (see Section 4). The *high-level* insights and used tools are described below.

Coding theory: Error correction code is used here to reduce the communication complexity in a way that the distributed processors exchange the encoded ("compressed") information symbols but not the initial ℓ-bit messages, where the encoded information symbol is a projection of a message on an encoding vector, e.g., $\boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_1$ is a projection of a message $\bar{\boldsymbol{w}}_1$ on an encoding vector $\boldsymbol{h}_i$ (see Fig. 2-(a)). The challenge is that the encoded symbols could not reveal enough information about the original messages, which might lead to some illusions at distributed processors and result in an inconsistent consensus. For example, as shown in Fig. 2-(a), when Processor i having an initial message as $\bar{\boldsymbol{w}}_1$ sends out an encoded symbol $\boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_1$, this encoded symbol might be expressed as $\boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_2$ if $\bar{\boldsymbol{w}}_1 - \bar{\boldsymbol{w}}_2$ is in the null space of $\boldsymbol{h}_i$. In this case, it creates an illusion at other processors that Processor i might initially hold the message as $\bar{\boldsymbol{w}}_2$ but not $\bar{\boldsymbol{w}}_1$. Our goal is to control the number of the aforementioned illusions to be small in the designed protocol and finally remove those illusions, by using graph theory and linear algebra.

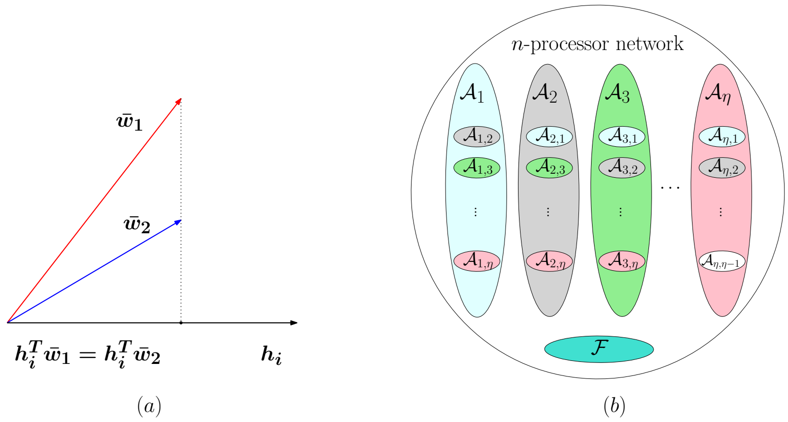


Figure 2 (a) Two vectors $\bar{\boldsymbol{w}}_1$ and $\bar{\boldsymbol{w}}_2$ have the same projection on an encoding vector $\boldsymbol{h}_i$. (b) An illustration of n-processor network. Group $\mathcal{F}$ denotes the indices of all dishonest processors. Group $\mathcal{A}_l$ denotes the indices of honest processors whose initial messages are all equal to a value $\bar{\boldsymbol{w}}_l$, for some $\bar{\boldsymbol{w}}_l$ and for $l \in [1:\eta]$. $\mathcal{A}_{l,j}$ denotes a subset of $\mathcal{A}_l$ such that $\boldsymbol{h}_i^\intercal\bar{\boldsymbol{w}}_l = \boldsymbol{h}_i^\intercal\bar{\boldsymbol{w}}_j, \forall i \in \mathcal{A}_{l,j}$, which means that the encoded information symbols sent from the processors in $\mathcal{A}_{l,j}$ seem to be projected from the value $\bar{\boldsymbol{w}}_j$ but actually projected from the value $\bar{\boldsymbol{w}}_l$, for $j \neq l, j, l \in [1:\eta]$.

Graph theory and linear algebra: To control the number of the aforementioned illusions to be small, we first classify the n-processor network into different groups. As shown in Fig. 2-(b), Group $\mathcal{A}_l$ denotes a set of honest processors holding the same value of initial messages, i.e., $\bar{\boldsymbol{w}}_l$, for $l \in [1:\eta]$, while $\mathcal{A}_{l,j}$ denotes a subset of $\mathcal{A}_l$ such that the encoded symbols sent from $\mathcal{A}_{l,j}$ seem to be projected from the value $\bar{\boldsymbol{w}}_j$, but actually projected from the value $\bar{\boldsymbol{w}}_l$. Therefore, the encoded information symbols sent from $\cup_{l=1,l\neq j}^{\eta}\mathcal{A}_{l,j}$ seem to be projected from the value $\bar{\boldsymbol{w}}_j$, but actually not. The idea of our design is to control the size of $\cup_{l=1,l\neq j}^{\eta}\mathcal{A}_{l,j}$ to be small for any j. Specifically, in our protocol the parameter η is controlled to be small by using graph theory, while the size of each $\mathcal{A}_{l,j}$ is controlled to be bounded by using linear algebra. For the use of graph theory, our approach is to map the n-node network into a specific type of graph, with one graph example depicted in Fig. 3, and bound the value of η based on this defined graph such that η cannot be more than 2 at the end of the second phase (see Lemma 22 and Lemma 18). For the use of linear algebra, our approach is to construct a set of linear equations based on the constraints related to $\mathcal{A}_{l,j}$, as well as the encoding matrix property, and then bound the size of $\mathcal{A}_{l,j}$ (see Lemma 21).

Coding has been used previously as an exciting approach in network communication and cooperative data exchange for improving throughput and tolerating attacks or failures [40, 37, 24, 10, 31, 34, 36, 38, 63, 32, 29, 62, 61, 17]. However, one common assumption in those previous works is that the source of the data is always *fault-free*, i.e., the source node is always honest or trustworthy, or the initial messages of the honest nodes are consistent and generated from the trustworthy source node. This is very different from the BA and BB problems considered here, in which the leader (or the source node) can be dishonest and the original messages of the distributed nodes can be controlled by the Byzantine adversary. One of the main difficulties of the BA and BB problems lies in the unknown knowledge about the leader (honest or dishonest) and about initial messages (consistent or inconsistent).

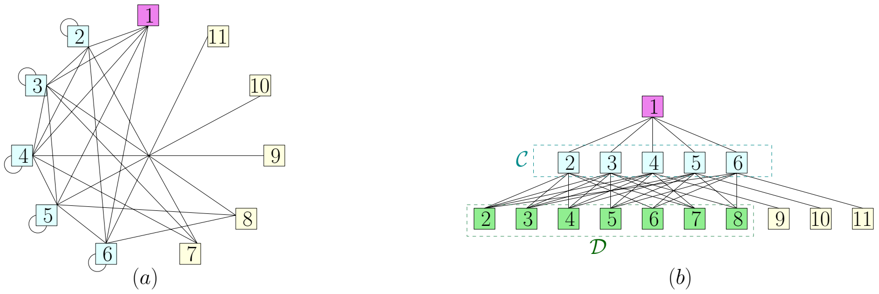


Figure 3 (a) One example of a type of graph, where each vertex in $\mathcal{C} = \{2, 3, 4, 5, 6\}$ is connected with at least 6 edges: one of the edges is connected to vertex $i^\star = 1$ and the rest are connected to the vertices in $\mathcal{D} = \{2, 3, 4, 5, 6, 7, 8\}$. (b) A three-layer representation of the graph in (a).

2 System models

In the BA problem, n processors want to reach agreement on an ℓ-bit message (or value), but up to t of the processors are dishonest (or faulty). Processor i holds an ℓ-bit initial message $\boldsymbol{w}_i$, $\forall i \in [1 : n]$. To solve this BA problem, a designed protocol needs to satisfy the *termination*, *consistency* and *validity* conditions mentioned in the previous section. We consider the synchronous BA, in which every two processors are connected via a reliable and private communication channel, and the messages sent on a channel are guaranteed to reach to the destination on time. We assume that a Byzantine adversary takes full control over the dishonest processors and has complete knowledge of the state of the other processors, including the ℓ-bit initial messages.

As mentioned, a protocol that satisfies the termination, consistency and validity conditions in all executions is said to be *error free*. A protocol that doesn't rely on the cryptographic technique such as signatures, hashing, authentication and secret sharing is said to be *signature free*. A protocol that is secure (satisfying the termination, consistency and validity conditions) against computationally unbounded adversary is said to be *information-theoretic secure*. In the BB problem, the validity condition requires that, if the leader is honest then all honest processors should agree on the message sent by the leader. Other definitions follow similarly from that of the BA problem.

3 Main results

The main results of this work are summarized in the following theorems.

▶ **Theorem 1** (BA problem). *The proposed COOL is an error-free signature-free information-theoretic-secure multi-valued BA protocol that achieves the consensus on an ℓ-bit message with optimal resilience, asymptotically optimal round complexity, and asymptotically optimal communication complexity when $\ell \geq t \log t$, simultaneously.*

Proof. The description of the proposed COOL is provided in Section 4. The proposed COOL achieves the consensus on an ℓ-bit message with resilience of $n \geq 3t + 1$ (optimal), round complexity of $O(t)$ rounds (asymptotically optimal), and communication complexity of $O(\max\{n\ell, nt \log t\})$ bits (asymptotically optimal when $\ell \geq t \log t$), simultaneously. Note that, for any error-free BA protocol, the known lower bounds on resilience, round complexity and communication complexity are $3t+1$ (cf. [52, 39]), $t+1$ (cf. [22, 20]), and $\Omega(\max\{n\ell, nt\})$ (cf. [19, 23]), respectively. ◀

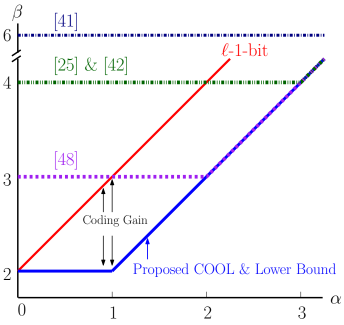


Figure 4 Communication complexity exponent β vs. message size exponent α of the proposed COOL, the protocols in [41, 25, 42, 48], and ℓ-1-bit scheme, focusing on the case with $\delta = 1$.

▶ **Theorem 2** (BB problem). *The proposed adapted COOL is an error-free signature-free information-theoretic-secure multi-valued BB protocol that achieves the consensus on an ℓ-bit message with optimal resilience, asymptotically optimal round complexity, and asymptotically optimal communication complexity when $\ell \geq t \log t$, simultaneously.*

Proof. The proposed COOL designed for the BA setting can be adapted into the BB setting, which achieves the consensus on an ℓ-bit message with the same performance of resilience, round complexity and communication complexity as in the BA setting. Note that the known lower bounds on resilience, round complexity and communication complexity for error-free BA protocols, can also be applied to any error-free BB protocols. The description of adapted COOL for the BB setting is provided in Appendix A. ◀

For an error-free BA (or BB) protocol, we define the notions of *communication complexity exponent*, *message size exponent* and *faulty size exponent* as $\beta(\alpha,\delta) \triangleq \lim_{n\to\infty} \frac{\log b(n,\delta,\alpha)}{\log n}$, $\alpha \triangleq \lim_{n\to\infty} \frac{\log \ell}{\log n}$, and $\delta \triangleq \lim_{n\to\infty} \frac{\log t}{\log n}$, respectively. Intuitively, β (resp. α and δ) captures the exponent of communication complexity b (resp. message size ℓ and faulty size t) with n as the base, when n is large. In this work we characterize the *optimal* communication complexity exponent $\beta^*(\alpha,\delta)$ achievable by *any* error-free BA protocol when $n \geq 3t+1$.

▶ **Theorem 3** (communication complexity exponent). *The optimal communication complexity exponent $\beta^*(\alpha,\delta)$ achievable by* any *error-free BA (or BB) protocol when $n \geq 3t+1$, is*

$$\beta^*(\alpha,\delta) = \max\{1+\alpha, 1+\delta\}. \tag{1}$$

Proof. Based on the known lower bound, the optimal communication complexity exponent is lower bounded by $\beta^*(\alpha,\delta) \geq \lim_{n\to\infty} \frac{\log \Omega(\max\{n\ell, nt\})}{\log n} = \max\{1+\alpha, 1+\delta\}$. This lower bound is achievable by the proposed COOL, as the communication complexity exponent of COOL, denoted by $\beta^{[cool]}$, is $\beta^{[cool]}(\alpha,\delta) = \lim_{n\to\infty} \frac{\log O(\max\{n\ell, nt\log t\})}{\log n} = \max\{1+\alpha, 1+\delta\}$. ◀

As shown in Fig. 4, the proposed COOL achieves the *optimal* communication complexity exponent. Compared to the protocols in [41, 25, 42, 48], COOL provides additive gains up to 4, 2, 2, 1, respectively, in terms of communication complexity exponent. Compared to ℓ-1-bit scheme without coding, COOL provides a communication complexity exponent gain of 1 for any $\alpha \geq 1$, which can be considered as a coding gain resulted from carefully-crafted coding.

4 COOL: coded Byzantine agreement protocol

This section describes the proposed COOL: **co**ded Byzantine agreement protoc**ol**. Error correction code is used in COOL. The (n, k) Reed-Solomon error correction code encodes k data symbols from Galois Field $GF(2^c)$ into a codeword consisting of n symbols from $GF(2^c)$, for $n \leq 2^c - 1$ (cf. [55])[1]. We can use c bits to represent each symbol from $GF(2^c)$, which implies that a vector consisting of k symbols from $GF(2^c)$ can be represented using kc bits of data. The error correction code can be constructed by *Lagrange polynomial interpolation*. An (n, k) error correction code can correct up to $\lfloor \frac{n-k}{2} \rfloor$ errors by applying some efficient decoding algorithms for Reed-Solomon code, such as, Berlekamp-Welch algorithm and Euclid's algorithm[56, 6, 55].

In the proposed COOL, at first the parameters k and c are designed as

$$k \triangleq \left\lfloor \frac{t}{5} \right\rfloor + 1, \quad c \triangleq \left\lceil \frac{\max\{\ell, \ (t/5+1)\cdot \log(n+1)\}}{k} \right\rceil. \tag{2}$$

As will be shown later, our design of k and c as above is one of the key elements in COOL, which guarantees that the proposed COOL satisfies the termination, consistency and validity conditions. With the above values of k and c, it holds true that the condition of the Reed-Solomon error correction code, i.e., $n \leq 2^c - 1$, is satisfied. The (n, k) Reed-Solomon error correction code is used to encode the ℓ-bit initial message $\boldsymbol{w}_i$, $\forall i \in [1:n]$. When ℓ is less than kc bits, the ℓ-bit message $\boldsymbol{w}_i$ will be first extended to a kc-bit data by adding $(kc - \ell)$ bits of redundant zeros (zero padding). The proposed COOL will work as long as $t \leq \frac{n-1}{3}$. In the description of the proposed COOL, t is considered such that $t \leq \frac{n-1}{3}$ and $t = \Omega(n)$. Later on we will discuss the case when t is relatively small compared to n.

The proposed COOL consists of at most four phases (see Fig. 1), which are described in the following sub-sections. The proposed COOL is also described in Algorithm 1 later. Let us first define $\boldsymbol{w}^{(i)}$ as the updated message at Processor i, $i \in [1:n]$. $\boldsymbol{w}^{(i)}$ can be updated via decoding, or via comparing its own information and the obtained information. In this proposed protocol, the decoding is required at Phase 4 only. The value of $\boldsymbol{w}^{(i)}$ is initially set as $\boldsymbol{w}^{(i)} = \boldsymbol{w}_i$, $i \in [1:n]$.

4.1 Phase 1: exchange compressed information and update message

Phase 1 has three steps. The idea is to exchange "compressed" information and learn it.

1) Exchange compressed information: Processor i, $i \in [1:n]$, first encodes its ℓ-bit initial message $\boldsymbol{w}_i$ into ℓ/k-bit symbols as

$$y_j^{(i)} \triangleq \boldsymbol{h}_j^{\intercal} \boldsymbol{w}_i, \quad j \in [1:n] \tag{3}$$

where $\boldsymbol{h}_j$ is defined as $\boldsymbol{h}_j \triangleq [h_{j,1}, h_{j,2}, \cdots, h_{j,k}]^{\intercal}$ and $h_{j,m} \triangleq \prod_{\substack{p=1 \\ p \neq m}}^{k} \frac{j-p}{m-p}$, $m \in [1:k]$. Then, Processor i sends coded symbols $(y_j^{(i)}, y_i^{(i)})$ to Processor j for $i, j \in [1:n], j \neq i$.

2) Update information: Processor i, $i \in [1:n]$, compares the observation $(y_i^{(j)}, y_j^{(j)})$ received from Processor j with its observation $(y_i^{(i)}, y_j^{(i)})$ and sets a binary indicator for the link between Processor i and Processor j, denoted by $\mathrm{u}_i(j)$, as

$$\mathrm{u}_i(j) = \begin{cases} 1 & \text{if } \ (y_i^{(j)}, y_j^{(j)}) = (y_i^{(i)}, y_j^{(i)}) \\ 0 & \text{else} \end{cases} \tag{4}$$

[1] For extended Reed-Solomon codes, the constraint can be relaxed to $n \leq 2^c + 2$ in some cases.

for $j \in [1:n]$. $\mathrm{u}_i(j)$ can be considered as a *link indicator* for Processor i and Processor j. The value of $\mathrm{u}_i(j) = 0$ reveals that Processor i and Processor j have mismatched messages, i.e., $\boldsymbol{w}^{(i)} \neq \boldsymbol{w}^{(j)}$. However, the value of $\mathrm{u}_i(j) = 1$ does *not* mean that Processor i and Processor j have matched messages; it just means that Processor i and Processor j share a common information at a certain degree, i.e., $(y_i^{(j)}, y_j^{(j)}) = (y_i^{(i)}, y_j^{(i)})$. When $\mathrm{u}_i(j) = 0$, the observation of $(y_i^{(j)}, y_j^{(j)})$ received from Processor j is considered as a *mismatched observation* at Processor i. In this step Processor i, $i \in [1:n]$, checks if its own initial message successfully matches the majority of other processors' initial messages, by counting the number of mismatched observations. Specifically, Processor i sets a binary *success indicator*, denoted by s_i, as

$$\mathrm{s}_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{n} \mathrm{u}_i(j) \geq n-t \\ 0 & \text{else .} \end{cases} \tag{5}$$

The event of $\mathrm{s}_i = 0$ means that the number of mismatched observations is more than t, which implies that the initial message of Processor i doesn't match the majority of other processors' initial messages. If $\mathrm{s}_i = 0$, Processor i updates the message as $\boldsymbol{w}^{(i)} = \phi$ (a default value), else keeps the original value of $\boldsymbol{w}^{(i)}$.

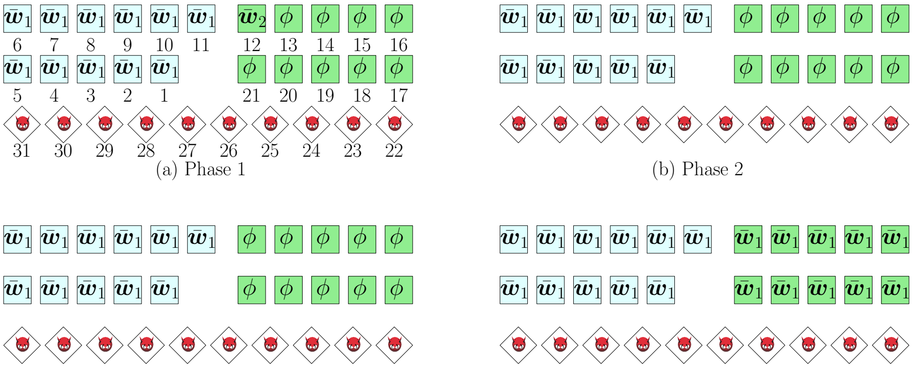
3) Exchange success indicators: Processor i, $i \in [1:n]$, sends the binary value of success indicator s_i to all other processors. Based on the received success indicators $\{\mathrm{s}_i\}_{i=1}^{n}$, each processor creates the following two sets:

$$\mathcal{S}_1 \triangleq \{i : \mathrm{s}_i = 1, i \in [1:n]\}, \quad \mathcal{S}_0 \triangleq \{i : \mathrm{s}_i = 0, i \in [1:n]\}. \tag{6}$$

Note that different processors might have different views on $\mathcal{S}_1$ and $\mathcal{S}_0$, due to the inconsistent information possibly sent from dishonest processors.

▶ Remark 4. *Since $y_j^{(i)}$ defined in* (3) *has only c bits, the total communication complexity among the network for the first step of Phase 1, denoted by b_1, is $b_1 = 2cn(n-1)$ bits. If Processor i, $i \in [1:n]$, sends the whole message $\boldsymbol{w}_i$ to other processors, then the total communication complexity would be $\ell n(n-1)$ bits. Compared to the whole message $\boldsymbol{w}_i$, the value of $y_j^{(i)}$ can be considered as a compressed information. By exchanging compressed information, instead of whole messages, the communication complexity is significantly reduced in this step. Since the success indicator s_i has only 1 bit, the communication complexity for the third step of Phase 1, denoted by b_2, is $b_2 = n(n-1)$ bits.*

▶ Remark 5. *In Phase 1, exchanging "compressed" information reduces the communication complexity, however, it also creates some potential issues due to the lack of full original information. Fig. 5 describes an example with three disjoint groups in n-processor network: Group $\mathcal{F}$, Group $\mathcal{A}_1$ and Group $\mathcal{A}_2$, where Group $\mathcal{A}_i$ is a set of honest processors holding the same initial message $\bar{\boldsymbol{w}}_i$, given $|\mathcal{A}_1| = t+1$, $|\mathcal{A}_2| = t$, and $\bar{\boldsymbol{w}}_1 \neq \bar{\boldsymbol{w}}_2$. To attack the protocol, in Phase 1 each dishonest processor could send inconsistent information to two different groups of honest processors, that is, sending symbols $(\boldsymbol{h}_j^{\mathsf{T}}\bar{\boldsymbol{w}}_1, \boldsymbol{h}_i^{\mathsf{T}}\bar{\boldsymbol{w}}_1)$ to Processor j for $j \in \mathcal{A}_1$ and sending different symbols $(\boldsymbol{h}_{j'}^{\mathsf{T}}\bar{\boldsymbol{w}}_2, \boldsymbol{h}_i^{\mathsf{T}}\bar{\boldsymbol{w}}_2)$ to Processor j' for $j' \in \mathcal{A}_2$, respectively, for $i \in \mathcal{F}$. Due to this inconsistent information, together with the condition of $\boldsymbol{h}_1^{\mathsf{T}}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_1^{\mathsf{T}}\bar{\boldsymbol{w}}_2$ and $\boldsymbol{h}_{12}^{\mathsf{T}}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_{12}^{\mathsf{T}}\bar{\boldsymbol{w}}_2$, the honest processors from different groups consequently have different updated messages, which might lead to inconsistent consensus outputs. In Phase 1 Processor i, $i \in [13:21] \subset \mathcal{A}_2$, sets $\mathrm{u}_i(j) = 0, \forall j \in [1:11]$ and $\mathrm{s}_i = 0$, from which it identifies that its initial message doesn't match the majority of other processors' initial messages and then updates its message as $\boldsymbol{w}^{(i)} = \phi$. However, Processor* 12 *still thinks that its initial message does match the majority of other processors' initial messages because of the condition*



(a) Phase 1

(b) Phase 2

(c) Phase 3

(d) Phase 4

Figure 5 Illustration of COOL for an example with $(t = 10, n = 31)$. The number under the node indicates the identity. The value inside the ith node denotes the value of updated message $\boldsymbol{w}^{(i)}$ at the end of the corresponding phase. Here $\mathcal{A}_1 = [1 : 11]$, $\mathcal{A}_2 = [12 : 21]$, $\mathcal{F} = [22 : 31]$, $\boldsymbol{w}_i = \bar{\boldsymbol{w}}_1, \forall i \in \mathcal{A}_1$ and $\boldsymbol{w}_{i'} = \bar{\boldsymbol{w}}_2, \forall i' \in \mathcal{A}_2$. It is assumed that $\boldsymbol{h}_1^{\intercal}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_1^{\intercal}\bar{\boldsymbol{w}}_2$ and $\boldsymbol{h}_{12}^{\intercal}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_{12}^{\intercal}\bar{\boldsymbol{w}}_2$, for $\bar{\boldsymbol{w}}_1 \neq \bar{\boldsymbol{w}}_2$. All honest processors eventually make the same consensus output.

$\boldsymbol{h}_1^{\intercal}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_1^{\intercal}\bar{\boldsymbol{w}}_2$ *and* $\boldsymbol{h}_{12}^{\intercal}\bar{\boldsymbol{w}}_1 = \boldsymbol{h}_{12}^{\intercal}\bar{\boldsymbol{w}}_2$. *This condition implies that* $(y_{12}^{(j)}, y_j^{(j)}) = (y_{12}^{(12)}, y_j^{(12)})$ *for any* $j \in \{1\} \cup \mathcal{A}_2$ *from the view of Processor* 12, *and hence results in a "wrong" (i.e., mismatched) output of* $\mathrm{s}_{12} = 1$ *at Processor* 12. *In the next phases the effort is to detect errors (mismatched information), mask errors, and identify "trusted" information.*

4.2 Phase 2: mask errors, and update success indicator and message

Phase 2 has three steps. The goal is to mask errors from the honest processors.

1) Mask errors identified in the previous phase: Processor i, $i \in \mathcal{S}_1$, sets

$$\mathrm{u}_i(j) = 0, \ \forall j \in \mathcal{S}_0. \tag{7}$$

2) Update and send success indicator: Processor i, $i \in \mathcal{S}_1$, updates s_i as in (5) using updated values of $\{\mathrm{u}_i(1), \cdots, \mathrm{u}_i(n)\}$. If the updated value of success indicator is $\mathrm{s}_i = 0$, then Processor i sends the updated success indicator of $\mathrm{s}_i = 0$ to others and updates the message as $\boldsymbol{w}^{(i)} = \phi$.

3) Update $\mathcal{S}_1$ *and* $\mathcal{S}_0$*:* Processor i, $i \in [1 : n]$, updates the sets of $\mathcal{S}_1$ and $\mathcal{S}_0$ as in (6) based on the newly received success indicators $\{\mathrm{s}_i\}_{i=1}^{n}$.

▶ Remark 6. *Since the success indicator* s_i *has only* 1 *bit, the total communication complexity for the second step of Phase 2, denoted by* b_3, *is bounded by* $b_3 \leq n(n-1)$ *bits.*

▶ Remark 7. *The idea of Phase 2 is to mask errors from honest processors whose initial messages don't match the majority of other processors' initial messages, but could not be detected out in Phase 1. For the example in Fig. 5, at the second step of Phase 2, Processor* 12 *updates the message as* $\boldsymbol{w}^{(12)} = \phi$. *This is because at Step 2 of Phase 2, from the view of Processor* 12, *the number of mismatched observations is at least* 19, *since* $\mathrm{u}_{12}(j) = 0, \forall j \in [2 : 11] \cup [13 : 21]$ *based on the updated information in* (7).

4.3 Phase 3: mask errors, update information, and vote

Phase 3 has five steps. The goal is to mask the rest of errors from the honest processors, and then vote for going to next phase or stopping in this phase.

1) Mask errors identified in the previous phase: Processor i, $i \in \mathcal{S}_1$, sets

$$\mathrm{u}_i(j) = 0, \ \forall j \in \mathcal{S}_0. \tag{8}$$

2) Update and send success indicator: Processor i, $i \in \mathcal{S}_1$, updates s_i as in (5) using updated values of $\{\mathrm{u}_i(1), \cdots, \mathrm{u}_i(n)\}$. If the updated value of success indicator is $\mathrm{s}_i = 0$, then Processor i sends $\mathrm{s}_i = 0$ to others and updates the message as $\boldsymbol{w}^{(i)} = \phi$.

3) Update $\mathcal{S}_1$ and $\mathcal{S}_0$: Processor i, $i \in [1:n]$, updates the sets of $\mathcal{S}_1$ and $\mathcal{S}_0$ as in (6) based on the newly received success indicators $\{\mathrm{s}_i\}_{i=1}^n$.

4) Vote: Processor i, $i \in [1:n]$, sets a binary vote as

$$\mathrm{v}_i = \begin{cases} 1 & \text{if } \sum_{j=1}^n \mathrm{s}_j \geq 2t+1 \\ 0 & \text{else} . \end{cases} \tag{9}$$

The indicator v_i can be considered as a vote for going to next phase or stopping in this phase.

5) One-bit consensus on the n votes: In this step the system runs one-bit consensus [7, 16] on the n votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ from all processors. If the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 1, then every honest processor *goes to next phase*, else every honest processor sets $\boldsymbol{w}^{(i)} = \phi$ and considers it as a final consensus and *stops here.*

► Remark 8. *Since* s_i *has only* 1 *bit, the total communication complexity for the second step of Phase 3, denoted by b_4, is bounded by $b_4 \leq n(n-1)$ bits.*

► Remark 9. *Since the system runs the one-bit consensus from [7, 16], the total communication complexity for the last step of Phase 3, denoted by b_5, is $b_5 = O(nt)$ bits, while the round complexity of this step is $O(t)$ rounds, which dominates the round complexity of COOL.*

► Remark 10. *The goal of the first two steps of Phase 3 is to mask the remaining errors from honest processors. As shown in Lemma 17 later, it is guaranteed that at the end of Phase 3 there exists* at most *1 group of honest processors, where the honest processors within this group have the same* non-empty *updated message (like $\mathcal{A}_1$ in Fig. 5).*

4.4 Phase 4: identify trusted information and make consensus

This phase is taken place only when one-bit consensus of $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 1.

1) Update information with majority rule: Processor i, $i \in \mathcal{S}_0$, updates $y_i^{(i)}$ as

$$y_i^{(i)} \leftarrow \mathrm{Majority}(\{y_i^{(j)} : j \in \mathcal{S}_1\}) \tag{10}$$

where the symbols of $y_i^{(j)}$ were received in Phase 1. Majority($\bullet$) is a function that returns the most frequent value in the list, based on majority rule. For example, $\mathrm{Majority}(1, 2, 2) = 2$.

2) Broadcast updated information: Processor i, $i \in \mathcal{S}_0$, sends the updated value of $y_i^{(i)}$ to Processor j, $\forall j \in \mathcal{S}_0, j \neq i$.

3) Decode the message: Processor i, $i \in \mathcal{S}_0$, decodes its message using the observations $\{y_1^{(1)}, y_2^{(2)}, \cdots, y_n^{(n)}\}$, where $\{y_j^{(j)} : j \in \mathcal{S}_0\}$ were updated and received in this phase and $\{y_j^{(j)} : j \in \mathcal{S}_1\}$ were received in the Phase 1. The value of $\boldsymbol{w}^{(i)}$ is updated as the decoded message at Processor i, $i \in \mathcal{S}_0$. For Processor i, $i \in \mathcal{S}_1$, it keeps the original value of $\boldsymbol{w}^{(i)}$.

4) Stop: Processor i, $i \in [1:n]$, outputs consensus as the message $\boldsymbol{w}^{(i)}$ and stops.

► Remark 11. *Since $y_i^{(i)}$ has only c bits, the total communication complexity for the second step of Phase 4, denoted by b_6, is upper bounded by $b_6 \leq cn(n-1)$ bits.*

▶ Remark 12. *The idea of Phase 4 is to identify "trusted" information from the set of honest processors whose initial messages match the majority of other processors' initial messages. In this way, the set of honest processors whose initial messages don't match the majority of other processors' initial messages could calibrate and update their information. For the example in Fig. 5, since the size of $\mathcal{A}_1$ is bigger than the size of $\mathcal{F}$, it guarantees that Group $\mathcal{A}_2$ could calibrate and update their information successfully in the first step of Phase 4, eventually making the same consensus output as Group $\mathcal{A}_1$.*

5 Provable performance of COOL

In this section we analyze the performance of COOL. At first we define some groups of processors in the n-processor network. For the ease of notation, let us use $\mathrm{s}_i^{[1]}$, $\mathrm{s}_i^{[2]}$ and $\mathrm{s}_i^{[3]}$ to denote the values of s_i updated in Phase 1, Phase 2 and Phase 3, respectively. Similarly, let us use $\mathrm{u}_i^{[1]}(j)$, $\mathrm{u}_i^{[2]}(j)$ and $\mathrm{u}_i^{[3]}(j)$ to denote the values of $\mathrm{u}_i(j)$ updated in Phase 1, Phase 2 and Phase 3, respectively. Let us first define Group $\mathcal{F}$ as the indices of all of the dishonest processors. Without loss of generality, *in the analysis* we just focus on the case with t dishonest nodes, i.e., $|\mathcal{F}| = t$ (no matter how the dishonest nodes act). Note that when $|\mathcal{F}| = t'$ for some $t' < t$, this case is indistinguishable from the case with $|\mathcal{F}| = t$ in which $t - t'$ out of t dishonest nodes act normally like honest nodes. Therefore, if a protocol is correct in the extreme case with $|\mathcal{F}| = t$, it is also correct in the case with $|\mathcal{F}| < t$.

Let us then define some groups of honest processors as

$$\mathcal{A}_l \triangleq \{i : \boldsymbol{w}_i = \bar{\boldsymbol{w}}_l,\ i \notin \mathcal{F},\ i \in [1:n]\}, \quad l \in [1:\eta] \tag{11}$$

$$\mathcal{A}_l^{[p]} \triangleq \{i : \mathrm{s}_i^{[p]} = 1, \boldsymbol{w}_i = \bar{\boldsymbol{w}}_l,\ i \notin \mathcal{F},\ i \in [1:n]\}, \quad l \in [1:\eta^{[p]}], \quad p \in \{1,2,3\} \tag{12}$$

$$\mathcal{B}^{[p]} \triangleq \{i : \mathrm{s}_i^{[p]} = 0,\ i \notin \mathcal{F},\ i \in [1:n]\}, \quad p \in \{1,2,3\} \tag{13}$$

for some different non-empty ℓ-bit values $\bar{\boldsymbol{w}}_1, \bar{\boldsymbol{w}}_2, \cdots, \bar{\boldsymbol{w}}_\eta$ and some non-negative integers $\eta, \eta^{[1]}, \eta^{[2]}, \eta^{[3]}$ such that $\eta^{[3]} \leq \eta^{[2]} \leq \eta^{[1]} \leq \eta$. Group $\mathcal{A}_l$ is a subset of honest processors who have the same value of initial messages. $\mathcal{A}_l^{[p]}$ (resp. $\mathcal{B}^{[p]}$) is a subset of honest processors who have the same non-empty (resp. empty) value of updated messages at the end of Phase p, for $p \in \{1,2,3\}$. As shown in Fig. 2-(a), two different messages might have the same projection on an encoding vector. With this motivation, Group $\mathcal{A}_l$ (and Group $\mathcal{A}_l^{[p]}$) can be divided into some possibly overlapping sub-groups defined as

$$\mathcal{A}_{l,j} \triangleq \{i :\ i \in \mathcal{A}_l,\ \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_l = \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_j\}, \quad j \neq l,\ j, l \in [1:\eta] \tag{14}$$

$$\mathcal{A}_{l,l} \triangleq \mathcal{A}_l \setminus \{\cup_{j=1, j\neq l}^{\eta} \mathcal{A}_{l,j}\}, \qquad l \in [1:\eta] \tag{15}$$

$$\mathcal{A}_{l,j}^{[p]} \triangleq \{i :\ i \in \mathcal{A}_l^{[p]},\ \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_l = \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_j\}, \quad j \neq l,\ j, l \in [1:\eta^{[p]}], \quad p \in \{1,2,3\} \tag{16}$$

$$\mathcal{A}_{l,l}^{[p]} \triangleq \mathcal{A}_l^{[p]} \setminus \{\cup_{j=1, j\neq l}^{\eta^{[p]}} \mathcal{A}_{l,j}^{[p]}\}, \qquad l \in [1:\eta^{[p]}], \quad p \in \{1,2,3\}. \tag{17}$$

The provable performance of COOL is summarized in the following Lemmas 13-16.

▶ **Lemma 13** (termination). *Given $n \geq 3t + 1$, all honest processors eventually output messages and terminate in COOL.*

Proof. Given $n \geq 3t + 1$, it is guaranteed in COOL that all honest processors eventually terminate together at the last step of Phase 3 or Phase 4. It is also guaranteed that every honest processor eventually outputs a message when it terminates. ◀

▶ **Lemma 14** (validity). *Given $n \geq 3t + 1$, if all honest processors have the same initial message, then at the end of COOL all honest processors agree on this initial message.*

Proof. When all honest processors have the same initial message, in the first three phases all honest processors will set their success indicators as ones and keep their updated messages exactly the same as the initial message, no matter what information sent by the dishonest processors. In this scenario, all honest processors will go to Phase 4 and eventually make the consensus outputs exactly the same as the initial message at the last step of Phase 4. ◀

▶ **Lemma 15** (consistency). *Given $n \geq 3t+1$, all honest processors reach the same agreement.*

Proof. We prove this lemma by considering each of the following two cases.

- Case (a): In Phase 3, the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 0.
- Case (b): In Phase 3, the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 1.

Analysis for Case (a): In Phase 3 of COOL, if the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 0, then each honest processor will set the updated message as ϕ and consider ϕ as a final consensus and stop here. In this case, all of the honest processors agree on the same message.

Analysis for Case (b): In Phase 3 of COOL, if the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 1, then all honest processors will go to Phase 4. In this case, at least one of the honest processors votes $\mathrm{v}_i = 1$, for some $i \notin \mathcal{F}$. Otherwise, all of the honest processors vote the same value such that $\mathrm{v}_i = 0$, $\forall i \notin \mathcal{F}$ and the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ should be 0, contradicting the condition of this case. In COOL, the condition of voting $\mathrm{v}_i = 1$ (see (9)) is that Processor i receives no less than $2t+1$ number of ones from n success indicators $\{\mathrm{s}_1^{[3]}, \mathrm{s}_2^{[3]}, \cdots, \mathrm{s}_n^{[3]}\}$, i.e., $\sum_{j=1}^{n} \mathrm{s}_j^{[3]} \geq 2t+1$. Since t dishonest processors might send the success indicators as ones, the above outcome implies that

$$\sum_{j \in \cup_{l=1}^{\eta^{[3]}} \mathcal{A}_l^{[3]}} \mathrm{s}_j^{[3]} \geq t+1. \tag{18}$$

Lemma 17 (see Section 5.1) reveals that at the end of Phase 3 there exists *at most* 1 group of honest processors, where the honest processors in this group have the same non-empty updated message, that is, $\eta^{[3]} \leq 1$. Then, for this Case (b), the conclusions in (18) and Lemma 17 imply that at the end of Phase 3 there exists *exactly* 1 group of *honest* processors with group size bigger than or equal to $t+1$, where the honest processors within this group have the same non-empty updated message. In other words, for this Case (b), we have

$$\eta^{[3]} = 1, \quad |\mathcal{A}_1^{[3]}| \geq t+1, \quad \boldsymbol{w}_i = \bar{\boldsymbol{w}}_1, \quad \forall i \in \mathcal{A}_1^{[3]} \tag{19}$$

by following from (18) and Lemma 17, as well as the definition of $\mathcal{A}_l^{[3]}$.

In Phase 4 Processor i, $i \in \mathcal{S}_0$, updates the value of $y_i^{(i)}$ as $y_i^{(i)} \leftarrow \mathrm{Majority}(\{y_i^{(j)} : j \in \mathcal{S}_1\})$ based on the majority rule, where the symbols of $y_i^{(j)}$ were received in Phase 1. In this step it is true that $\mathcal{A}_1^{[3]} \subseteq \mathcal{S}_1$ and $|\mathcal{A}_1^{[3]}| > |\mathcal{F}|$ (see (19)), which guarantees that Processor i, $i \in \mathcal{S}_0 \setminus \mathcal{F}$, could use the majority rule to update the value of $y_i^{(i)}$ as $y_i^{(i)} \leftarrow \mathrm{Majority}(\{y_i^{(j)} : j \in \mathcal{S}_1\}) = \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_1$. At the end of this step, for any honest Processor i, $i \notin \mathcal{F}$, the value of $y_i^{(i)}$ becomes $y_i^{(i)} = \boldsymbol{h}_i^{\intercal}\bar{\boldsymbol{w}}_1$, which is encoded with $\bar{\boldsymbol{w}}_1$. In the second step of Phase 4, Processor i, $i \in \mathcal{S}_0$, sends the updated value of $y_i^{(i)}$ to Processor j, $\forall j \in \mathcal{S}_0, j \neq i$. After this, Processor i, $i \in \mathcal{S}_0$, decodes its message using the updated observations $\{y_1^{(1)}, y_2^{(2)}, \cdots, y_n^{(n)}\}$, where $n-t$ observations of which are guaranteed to be $y_j^{(j)} = \boldsymbol{h}_j^{\intercal}\bar{\boldsymbol{w}}_1, \forall j \notin \mathcal{F}$. Since the number of mismatched observations, i.e., the observations not encoded with the message $\bar{\boldsymbol{w}}_1$, is no more than t, then Processor i, $i \in \mathcal{S}_0 \setminus \mathcal{F}$, decodes its message and outputs $\boldsymbol{w}^{(i)} = \bar{\boldsymbol{w}}_1$. At the same time, Processor i, $i \in \mathcal{S}_1 \setminus \mathcal{F}$, outputs the original value as $\boldsymbol{w}^{(i)} = \bar{\boldsymbol{w}}_1$. Thus, all of the honest processors successfully agree on the same message, i.e., $\boldsymbol{w}^{(i)} = \bar{\boldsymbol{w}}_1$, $\forall i \notin \mathcal{F}$. With this we complete the proof of Lemma 15. ◀

▶ **Lemma 16** (complexity). *When $n \geq 3t+1$, COOL achieves the consensus on an ℓ-bit message with the communication complexity of $O(\max\{n\ell, nt\log t\})$ bits and $O(t)$ rounds.*

Proof. The total communication complexity of COOL, denoted by b, is computed as $b = \sum_{i=1}^{6} b_i = O(cn(n-1)+n^2) = O\Big(\Big\lceil \frac{\max\{\ell,\ (t/5+1)\cdot\log(n+1)\}}{\lfloor \frac{t}{5}\rfloor+1}\Big\rceil \cdot n(n-1)+n^2\Big) = O(\max\{\ell n^2/t, n^2\log n\})$ bits, where $b_1, b_2, \cdots, b_6$ are expressed in Remarks 4, 6, 8, 9 and 11, and the parameters c and k are defined in (2). In the description of the proposed protocol, t is considered such that $t \leq (n-1)/3$ and $t = \Omega(n)$. In this case, the total communication complexity computed as above can be rewritten in the form of $b = O(\max\{\ell n, nt\log t\})$ bits. For the case when t is relatively small, we show that COOL can be slightly modified to achieve the communication complexity as $b = O(\max\{\ell n, nt\log t\})$ bits (see the long version [14] for more details). The round complexity of COOL is dominated by the round complexity of the one-bit consensus in Phase 3, which is $O(t)$ rounds. ◀

From Lemmas 13-15, it reveals that given $n \geq 3t+1$ the termination, validity and consistency conditions are all satisfied in COOL in all executions (*error free*). Note that Lemmas 13-15 hold true without using any assumptions on the cryptographic technique (*signature-free*). Lemmas 13-15 also hold true even when the adversary, who takes full control over the dishonest processors, has unbounded computational power (*information-theoretic-secure*). The results of Lemmas 13-16 serve as the achievability proof of Theorem 1.

5.1 Some lemmas

Below we provide some lemmas that are used in the protocol analysis.

▶ **Lemma 17.** *Given $n \geq 3t+1$, at the end of Phase 3 of COOL there exists* at most *1 group of honest processors, where honest processors in this group have the same non-empty updated message, and honest processors outside this group have the same empty updated message.*

Proof. Based on (12), it is equivalent to prove $\eta^{[3]} \leq 1$. In the first step we prove that $\eta^{[2]} \leq 2$ (see Lemma 18 below). Clearly, it is true $\eta^{[3]} \leq 1$ when $\eta^{[2]} \leq 1$, using the fact that $\eta^{[3]} \leq \eta^{[2]}$ (see (11) and (12)). Then in the second step we prove that $\eta^{[3]} \leq 1$ when $\eta^{[2]} = 2$ (see Lemma 19). Thus, it is concluded that $\eta^{[3]} \leq 1$ for COOL with $n \geq 3t+1$. ◀

▶ **Lemma 18.** *For the proposed COOL with $n \geq 3t+1$, it holds true that $\eta^{[2]} \leq 2$.*

Proof. Proof by contradiction is used in this proof. Let us first assume that the claim in Lemma 18 is false. Specifically let us assume that

$$\eta^{[2]} \geq 3. \tag{20}$$

From Lemma 22 that will be shown later, we have $|\mathcal{A}_l| \geq n - 9t/4, l \in [1:\eta^{[2]}]$. By combining the above $\eta^{[2]}$ bounds together we have

$$\sum_{l=1}^{\eta^{[2]}} |\mathcal{A}_l| \geq \eta^{[2]}(n-9t/4) \geq 3(n-9t/4) = (n-t) + (2n-23t/4) > (n-t) \tag{21}$$

where the second inequality uses the assumption in (20); and the last inequality stems from the derivation that $2n - 23t/4 \geq 6t + 2 - 23t/4 > 0$ by using the condition of $n \geq 3t+1$. One can see that the conclusion in (21) contradicts with the identify of $\sum_{l=1}^{\eta^{[2]}} |\mathcal{A}_l| \leq \sum_{l=1}^{\eta} |\mathcal{A}_l| = n-t$, i.e., the total number of honest processors should be $n-t$. Therefore, the assumption in (20) leads to a contradiction and thus $\eta^{[2]}$ should be bounded by $\eta^{[2]} \leq 2$. ◀

▶ **Lemma 19.** *For COOL with $n \geq 3t+1$, it holds true that $\eta^{[3]} \leq 1$ when $\eta^{[2]} = 2$.*

Proof. Given $\eta^{[2]} = 2$, it is concluded from Lemma 20 (shown below) that $\eta^{[1]} = 2$. Furthermore, given $\eta^{[1]} = 2$, it is concluded from Lemma 23 (shown later) that $\eta^{[3]} \leq 1$. We then conclude that $\eta^{[3]} \leq 1$ when $\eta^{[2]} = 2$. ◀

▶ **Lemma 20.** *For the proposed COOL with $n \geq 3t+1$, it is true that $\eta^{[1]} = 2$ when $\eta^{[2]} = 2$.*

Proof. Proof by contradiction is also used here. Given $\eta^{[2]} = 2$, let us assume that $\eta^{[1]} > 2$. Given $\eta^{[2]} = 2$ and Lemma 22 (shown later), we have $|\mathcal{A}_l| \geq n - 9t/4$, $\forall l \in [1:2]$ and have

$$\sum_{l=3}^{\eta} |\mathcal{A}_l| = n - |\mathcal{F}| - |\mathcal{A}_1| - |\mathcal{A}_2| \leq n - t - 2(n - 9t/4) \leq t/2 - 1. \tag{22}$$

If $\eta^{[1]} > 2$, there exists an $i^\star \in \mathcal{A}_{l^\star}^{[1]} \subseteq \mathcal{A}_{l^\star}$ for $l^\star \geq 3$, such that $\mathrm{s}_{i^\star}^{[1]} = 1$ (see (12)) and that $\sum_{j \in \cup_{l=1}^{\eta} \mathcal{A}_l} \mathrm{u}_{i^\star}^{[1]}(j) \geq n - t - t$ (see (5)). Based on the definition of $\mathcal{A}_{l,j}$ in (14), it is true that $\mathrm{u}_{i^\star}^{[1]}(j) = 0$, $\forall j \in \{\cup_{l=1}^{\eta} \mathcal{A}_l\} \setminus \{\mathcal{A}_{l^\star} \cup \{\cup_{l=1, l \neq l^\star}^{\eta} \mathcal{A}_{l,l^\star}\}\}$. Therefore, the aforementioned inequality resulted from (5) can be rewritten as

$$\sum_{j \in \mathcal{A}_{l^\star} \cup \{\cup_{l=1, l \neq l^\star}^{\eta} \mathcal{A}_{l,l^\star}\}} \mathrm{u}_{i^\star}^{[1]}(j) \geq n - t - t. \tag{23}$$

On the other hand, the size of $\mathcal{A}_{l^\star} \cup \{\cup_{l=1, l \neq l^\star}^{\eta} \mathcal{A}_{l,l^\star}\}$ can be upper bounded by

$$\begin{aligned}
|\mathcal{A}_{l^\star} \cup \{\cup_{l=1, l \neq l^\star}^{\eta} \mathcal{A}_{l,l^\star}\}| &\leq |\mathcal{A}_{1,l^\star}| + |\mathcal{A}_{2,l^\star}| + \sum_{l=3}^{\eta} |\mathcal{A}_l| && (24)\\
&\leq |\mathcal{A}_{1,l^\star}| + |\mathcal{A}_{2,l^\star}| + t/2 - 1 && (25)\\
&\leq (k-1) + (k-1) + t/2 - 1 && (26)\\
&= 2(\lfloor t/5 \rfloor + 1 - 1) + t/2 - 1 && (27)\\
&< n - 2t && (28)
\end{aligned}$$

for $i^\star \in \mathcal{A}_{l^\star}^{[1]} \subseteq \mathcal{A}_{l^\star}$ and $l^\star \geq 3$, where (24) uses the fact that $\mathcal{A}_{l^\star} \cup \{\cup_{l=1, l \neq l^\star}^{\eta} \mathcal{A}_{l,l^\star}\} \subseteq \mathcal{A}_{1,l^\star} \cup \mathcal{A}_{2,l^\star} \cup \{\cup_{l=3}^{\eta} \mathcal{A}_l\}$; (25) is from (22); (26) stems from the result in Lemma 21 shown below; (27) uses the definition of k as in (2); (28) follows from the condition of $n \geq 3t+1$.

One can see that the conclusion in (28) contradicts with (23). Therefore, the assumption of $\eta^{[1]} > 2$ leads to a contradiction and thus $\eta^{[1]}$ should be $\eta^{[1]} = 2$, given $\eta^{[2]} = 2$. ◀

▶ **Lemma 21.** *For $\eta \geq \eta^{[1]} \geq 2$, the following inequalities hold true*

$$\begin{aligned}
|\mathcal{A}_{l,j}| + |\mathcal{A}_{j,l}| &< k, \quad \forall j \neq l, \ j, l \in [1:\eta] && (29)\\
|\mathcal{A}_{l,j}^{[1]}| + |\mathcal{A}_{j,l}^{[1]}| &< k, \quad \forall j \neq l, \ j, l \in [1:\eta^{[1]}] && (30)
\end{aligned}$$

where k is defined in (2)*.*

Proof. This proof borrows tool from linear algebra. Our approach is to construct a set of linear equations based on $\mathcal{A}_{l,j}$ and $\mathcal{A}_{j,l}$, as well as the full rank property of encoding matrix, and then bound the size of $\mathcal{A}_{l,j}$ and $\mathcal{A}_{j,l}$. Details are provided in the long version [14]. ◀

▶ **Lemma 22.** *When $\eta^{[2]} \geq 1$, it holds true that $|\mathcal{A}_l| \geq n - 9t/4$, for any $l \in [1:\eta^{[2]}]$.*

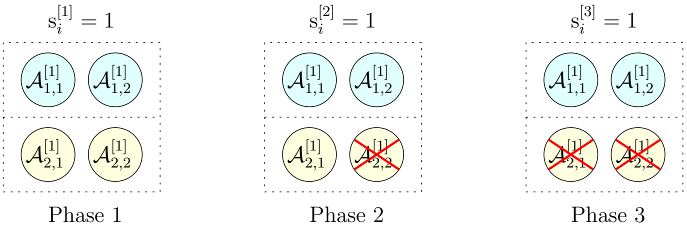

$s_i^{[1]} = 1$
$s_i^{[2]} = 1$
$s_i^{[3]} = 1$
$\mathcal{A}_{1,1}^{[1]}$
$\mathcal{A}_{1,2}^{[1]}$
$\mathcal{A}_{2,1}^{[1]}$
$\mathcal{A}_{2,2}^{[1]}$
Phase 1
Phase 2
Phase 3

Figure 6 An example with $\eta^{[3]} \leq 1$ given $\eta^{[1]} = 2$.

Proof. This proof borrows tool from graph theory. It consists of the following steps:

- Step (a): Transform the network into a graph that is within the family of graphs, for a fixed $i^\star$ in $\mathcal{A}_{l^\star}^{[2]}$ and $l^\star \in [1 : \eta^{[2]}]$. One graph example is depicted in Fig. 3.
- Step (b): Bound the size of a group of honest processors, denoted by $\mathcal{D}'$, using the result of a derived lemma based on the defined graph, i.e., $|\mathcal{D}'| \geq n - 9t/4 - 1$.
- Step (c): Argue that every processor in $\mathcal{D}'$ has the same initial message as Processor $i^\star$.
- Step (d): Conclude from Step (c) that $\mathcal{D}'$ is a subset of $\mathcal{A}_{l^\star}$, i.e., $\mathcal{D}' \cup \{i^\star\} \subseteq \mathcal{A}_{l^\star}$ and conclude that the size of $\mathcal{A}_{l^\star}$ is bounded by the number determined in Step (b), i.e., $|\mathcal{A}_{l^\star}| \geq |\mathcal{D}'| + 1 \geq n - 9t/4 - 1 + 1$, for $l^\star \in [1 : \eta^{[2]}]$.

More details are provided in the long version [14] due to the lack of space here. ◀

▶ **Lemma 23.** *For COOL with $n \geq 3t + 1$, if $\eta^{[1]} = 2$ then it holds true that $\eta^{[3]} \leq 1$.*

Proof. Given two groups of honest processors with two *non-empty* updated messages at the end of Phase 1 (i.e., $\eta^{[1]} = 2$), we prove that at least one group will be reduced to the one with *empty* updated message after error detecting and masking in Phases 2 and 3 (i.e., $\eta^{[3]} \leq 1$, see one example in Fig. 5). The high-level proof procedure is described in Fig. 6 for one scenario. Specifically, given $\eta^{[1]} = 2$, at the end of Phase 1 Groups $\mathcal{A}_1$ and $\mathcal{A}_2$ all have their success indicators as ones. After error detecting and masking in Phase 2 (resp. Phase 3), then Sub-group $\mathcal{A}_{2,2}$ (resp. $\mathcal{A}_{2,1}$) will update their success indicators as zeros. In this case, Group $\mathcal{A}_2$ will be reduced to the one with zero indicators at the end of Phase 3 (i.e., $\eta^{[3]} \leq 1$). Details are provided in the long version [14]. ◀

6 Conclusion and discussion

In this work we proposed COOL, a deterministic error-free signature-free BA protocol designed from coding theory, together with graph theory and linear algebra, with optimal or near optimal performance in resilience, round complexity, and communication complexity, simultaneously. The results reveal that coding is an effective approach for achieving the fundamental limits of Byzantine agreement and its variants. To see the advantages of coding, let us compare three schemes below.

Scheme with full data transmission (DT): The multi-valued consensus problems could be solved by invoking ℓ instances of 1-bit consensus in sequence (ℓ-1-bit scheme). Since distributed processors need to exchange the whole message data, it is not surprising that this scheme results in a high communication complexity (see Fig. 4).

Non-coding scheme with reduced DT: To reduce the communication complexity, one possible solution is to let distributed processors exchange only one piece of data, instead of the whole data. However, if the data is *uncoded*, this reduction in data exchange could possibly lead to a consensus error. This is because honest processors might not get enough information from others and hence this protocol is more vulnerable to the Byzantine attacks.

Coding scheme with reduced DT: If the data is *coded* with error correction code, the distributed processors could possibly be able to detect and correct errors, even with reduced data transmission. Hence, the coding scheme with reduced DT could be robust against attacks from dishonest processors and could be error free. One example of the coding scheme with reduced DT is the proposed COOL in the BA and BB settings. Table 2 provides some comparison between the above three schemes. We believe that the coding schemes could be applied to broader settings of distributed algorithms.

Table 2 Comparison of three consensus schemes.

Schemes	communication complexity	error free
Scheme with full DT	high	yes
Non-coding scheme with reduced DT	low	no
Coding scheme with reduced DT	low	yes

References

1 I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren. Efficient synchronous Byzantine consensus. Available on ArXiv: `arXiv:1704.02397`, 2017.

2 I. Abraham, D. Dolev, and J. Halpern. An almost-surely terminating polynomial protocol for asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 405–414, 2008.

3 S. Ahmed. A performance study of Hyperledger Fabric in a smart home and IoT environment, 2019. Available on https://www.duo.uio.no/handle/10852/70940.

4 Z. Amsden et al. The libra blockchain. *Libra White Paper*, 2019.

5 E. Androulaki et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

6 E. Berlekamp. Nonbinary BCH decoding (abstr.). *IEEE Trans. Inf. Theory*, 14(2):242–242, 1968.

7 P. Berman, J. Garay, and K. Perry. Bit optimal distributed consensus. *Computer Science*, pages 313–321, 1992.

8 V. Buterin. A next-generation smart contract and decentralized application platform. *Ethereum White Paper*, 2015.

9 C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2005.

10 N. Cai and R. Yeung. Network error correction, II: Lower bounds. *Communications in Information and Systems*, 6(1):37–54, 2006.

11 R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *25th Annual ACM Symposium on the Theory of Computing*, pages 42–51, 1993.

12 F. Casino, T. Dasaklis, and C. Patsakis. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics*, 36:55–81, March 2019.

13 B. Chan and E. Shi. Streamlet: Textbook streamlined blockchains. In *2nd ACM Conference on Advances in Financial Technologies(AFT)*, pages 1–11, October 2020.

14 J. Chen. Fundamental limits of Byzantine agreement. Available on ArXiv: `arXiv:10965`, 2020.

15 B. Chor and B. Coan. A simple and efficient randomized Byzantine agreement algorithm. *IEEE Transactions on Software Engineering*, 11(6):531–539, June 1985.

16 B. Coan and J. Welch. Modular construction of a Byzantine agreement protocol with optimal message bit complexity. *Information and Computation*, 97(1):61–85, March 1992.

17 T. Courtade and T. Halford. Coded cooperative data exchange for a secret key. *IEEE Trans. Inf. Theory*, 62(7):3785–3795, 2016.

18 M. Demir, M. Alalfi, O. Turetken, and A. Ferworn. Security smells in smart contracts. In *IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2019.

19 D. Dolev and R. Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985.

20 D. Dolev and H. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

21 P. Feldman and S. Micali. An optimal probabilistic protocol for synchronous Byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, August 1997.

22 M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

23 M. Fitzi and M. Hirt. Optimally efficient multi-valued Byzantine agreement. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–168, 2006.

24 C. Fragouli, D. Lun, M. Medard, and P. Pakzad. On feedback for network coding. In *2007 41st Annual Conference on Information Sciences and Systems*, March 2007.

25 C. Ganesh and A. Patra. Optimal extension protocols for Byzantine broadcast and agreement. In *Distributed Computing*, July 2020.

26 J. Garay and Y. Moses. Fully polynomial Byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

27 G. Goodson, J. Wylie, G. Ganger, and M. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks*, June 2004.

28 C. Gorenflo, S. Lee, L. Golab, and S. Keshav. FastFabric: Scaling Hyperledger Fabric to 20,000 transactions per second. In *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019.

29 W. Halbawi, T. Ho, H. Yao, and I. Duursma. Distributed reed-solomon codes for simple multiple access networks. In *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, 2014.

30 M. Hammoudeh, I. Ghafir, A. Bounceur, and T. Rawlinson. Continuous monitoring in mission-critical applications using the internet of things and blockchain. In *3rd International Conference on Future Networks and Distributed Systems*, July 2019.

31 T. Ho, B. Leong, R. Koetter, M. Medard, M. Effros, and D. Karger. Byzantine modification detection in multicast networks using randomized network coding. In *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, June 2004.

32 D. Huang and D. Medhi. A Byzantine resilient multi-path key establishment scheme and its robustness analysis for sensor networks. In *19th IEEE International Parallel and Distributed Processing Symposium*, April 2005.

33 Hyperledger. Hyperledger-fabricdocs documentation, 2020.

34 S. Jaggi, M. Langberg, S. Katti, T. Ho, D. Katabi, M. Medard, and M. Effros. Resilient network coding in the presence of Byzantine adversaries. *IEEE Trans. Inf. Theory*, 54(6):2596–2603, June 2008.

35 J. Katz and C. Koo. On expected constant-round protocols for Byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.

36 S. Kim, T. Ho, M. Effros, and S. Avestimehr. New results on network error correction: Capacities and upper bounds. In *Proc. Inf. Theory and App. Workshop (ITA)*, 2010.

37 R. Koetter and M. Medard. An algebraic approach to network coding. In *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, June 2001.

38 M. Krohn, M. Freedman, and D. Mazieres. On-the-fly verification of rateless erasure codes for efficient content distribution. In *IEEE Symposium on Security and Privacy (SP)*, May 2004.

39 L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982.

40 S. Li, R. Yeung, and N. Cai. Linear network coding. *IEEE Trans. Inf. Theory*, 49(2):371–381, February 2003.

41 G. Liang and N. Vaidya. Error-free multi-valued consensus with Byzantine failures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–20, June 2011.

42 A. Loveless, R. Dreslinski, and B. Kasikci. Optimal and error-free multi-valued Byzantine consensus through parallel execution. Available on : https://eprint.iacr.org/2020/322, 2020.

43 A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The Honey Badger of BFT protocols. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

44 B. Mohanta, D. Jena, S. Panda, and S. Sobhanayak. Blockchain technology: A survey on applications and security privacy challenges. *Internet of Things*, 8, 2019.

45 Y. Moses and O. Waarts. Coordinated traversal: $(t+1)$-round Byzantine agreement in polynomial time. *Journal of Algorithms*, 17(1):110–156, July 1994.

46 A. Mostéfaoui, H. Moumen, and M. Raynal. Signature-free asynchronous binary Byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of the ACM*, 62(4), 2015.

47 A. Mostéfaoui and M. Raynal. Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t < n/3$, $O(n^2)$ messages, and constant time. In *Proc. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO'15)*, July 2015.

48 K. Nayak, L. Ren, E. Shi, N. Vaidya, and Z. Xiang. Improved extension protocols for Byzantine broadcast and agreement. In *International Symposium on Distributed Computing (DISC)*, October 2020.

49 R. Pass and E. Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology – EUROCRYPT 2018. Lecture Notes in Computer Science*, volume 10821, pages 3–33, March 2018.

50 A. Patra. Error-free multi-valued broadcast and Byzantine agreement with optimal communication complexity. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 34–49, 2011.

51 A. Patra and P. Rangan. Communication optimal multi-valued asynchronous Byzantine agreement with optimal resilience. In *International Conference on Information Theoretic Security*, pages 206–226, 2011.

52 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

53 X. Qi, Z. Zhang, C. Jin, and A. Zhou. BFT-Store: Storage partition for permissioned blockchain via erasure coding. In *IEEE 36th International Conference on Data Engineering*, April 2020.

54 M. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, November 1983.

55 I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, June 1960.

56 R. Roth. *Introduction to coding theory.* Cambridge University Press, 2006.

57 E. Shi. Streamlined blockchains: A simple and elegant approach (a tutorial and survey). *Advances in Cryptology – ASIACRYPT 2019, Lecture Notes in Computer Science*, 11921:3–17, November 2019.

58 J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, May 2012.

59 J. Sousa, A. Bessani, and M. Vukolic. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018.

60 The Hyperledger White Paper Working Group. An introduction to Hyperledger. *Hyperledger White Paper*, 2018.

61 M. Yan and A. Sprintson. On error correcting algorithms for the cooperative data exchange problem. In *2014 International Symposium on Network Coding (NetCod)*, June 2014.

62 H. Yao and T. Ho. Distributed reed-solomon codes for simple multiple access networks, September 2015. US Patent 9,148,173.

63 Z. Yu, Y. Wei, B. Ramkumar, and Y. Guan. An efficient signature-based scheme for securing network coding against pollution attacks. In *IEEE International Conference on Computer Communications*, April 2008.

A The extension of COOL to the Byzantine broadcast problem

For the BB setting, at first the leader sends each processor an ℓ-bit message that can be considered as the initial message. Then, COOL is applied into this BB setting from this step to achieve the consistent and validated consensus, with similar performance as in BA setting.

B Algorithm of the proposed COOL protocol

Algorithm 1 : COOL protocol, code for Processor i, $i \in [1:n]$.

```
Initially set $\boldsymbol{w}^{(i)} = \boldsymbol{w}_i$.
Processor $i$ encodes its message into $n$ symbols as $y_j^{(i)} \triangleq \boldsymbol{h}_j^{\intercal}\boldsymbol{w}_i,\ j \in [1:n]$.
Phase 1
Processor $i$ sends $(y_j^{(i)}, y_i^{(i)})$ to Processor $j$, $\forall j \in [1:n], j \neq i$.
for $j = 1:n$ do
    if $((y_i^{(j)}, y_j^{(j)}) == (y_i^{(i)}, y_j^{(i)}))$ then
        Processor $i$ sets $\mathrm{u}_i(j) = 1$.
    else
        Processor $i$ sets $\mathrm{u}_i(j) = 0$.
if $(\sum_{j=1}^{n} \mathrm{u}_i(j) >= n - t)$ then
    Processor $i$ sets its success indicator as $\mathrm{s}_i = 1$.
else
    Processor $i$ sets $\mathrm{s}_i = 0$ and $\boldsymbol{w}^{(i)} = \phi$.
Processor $i$ sends the value of $\mathrm{s}_i$ to all other processors.
Processor $i$ creates sets $\mathcal{S}_p = \{j : \mathrm{s}_j = p, j \in [1:n]\}$, $p \in \{0,1\}$, from received $\{\mathrm{s}_j\}_{j=1}^{n}$.
Phase 2
if $(\mathrm{s}_i == 1)$ then
    Processor $i$ sets $\mathrm{u}_i(j) = 0, \forall j \in \mathcal{S}_0$.
    if $(\sum_{j=1}^{n} \mathrm{u}_i(j) < n - t)$ then
        Processor $i$ sets $\mathrm{s}_i = 0$ and $\boldsymbol{w}^{(i)} = \phi$.
        Processor $i$ sends the value of $\mathrm{s}_i$ to all other processors.
Processor $i$ updates $\mathcal{S}_0$ and $\mathcal{S}_1$ based on the newly received success indicators.
Phase 3
if $(\mathrm{s}_i == 1)$ then
    Processor $i$ sets $\mathrm{u}_i(j) = 0, \forall j \in \mathcal{S}_0$.
    if $(\sum_{j=1}^{n} \mathrm{u}_i(j) < n - t)$ then
        Processor $i$ sets $\mathrm{s}_i = 0$ and $\boldsymbol{w}^{(i)} = \phi$.
        Processor $i$ sends the value of $\mathrm{s}_i$ to all other processors.
Processor $i$ updates $\mathcal{S}_0$ and $\mathcal{S}_1$ based on the newly received success indicators.
if $(\sum_{j=1}^{n} \mathrm{s}_j >= 2t + 1)$ then
    Processor $i$ sets the binary vote as $\mathrm{v}_i = 1$.
else
    Processor $i$ sets the binary vote as $\mathrm{v}_i = 0$.
Processor $i$ runs the one-bit consensus with all other processors on votes $\{\mathrm{v}_j\}_j$ using one-bit consensus from [7, 16].
if (the consensus of the votes $\{\mathrm{v}_1, \mathrm{v}_2, \cdots, \mathrm{v}_n\}$ is 1) then
    Processor $i$ goes to next phase.
else
    Processor $i$ sets $\boldsymbol{w}^{(i)} = \phi$ and considers it as a final consensus and stops here.
Phase 4
if $(\mathrm{s}_i == 0)$ then
    Processor $i$ updates $y_i^{(i)} \leftarrow \mathrm{Majority}(\{y_i^{(j)} : \mathrm{s}_j = 1, j \in [1:n]\})$.
    Processor $i$ sends updated $y_i^{(i)}$ to Processor $j$, $\forall j \in \mathcal{S}_0, j \neq i$.
    Processor $i$ decodes message with new observations $\{y_1^{(1)}, \cdots, y_n^{(n)}\}$ and updates $\boldsymbol{w}^{(i)}$.
Processor $i$ outputs consensus as the updated message $\boldsymbol{w}^{(i)}$ and stops.
```

Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer

Shir Cohen ✉
Technion, Haifa, Israel

Idit Keidar ✉
Technion, Haifa, Israel

Abstract

We formalize Byzantine linearizability, a correctness condition that specifies whether a concurrent object with a sequential specification is resilient against Byzantine failures. Using this definition, we systematically study Byzantine-tolerant emulations of various objects from registers. We focus on three useful objects– reliable broadcast, atomic snapshot, and asset transfer. We prove that there exist n-process f-resilient Byzantine linearizable implementations of such objects from registers if and only if $f < \frac{n}{2}$.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases Byzantine linearizability, concurrent algorithms, snapshot, asset transfer

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.18

Related Version *Full Version*: https://arxiv.org/abs/2102.10597 [14]

1 Introduction

Over the last decade, cryptocurrencies have taken the world by storm. The idea of a decentralized bank, independent of personal motives has gained momentum, and cryptocurrencies like Bitcoin [23], Ethereum [25], and Diem [8] now play a big part in the world's economy. At the core of most of these currencies lies the asset transfer problem. In this problem, there are multiple accounts, operated by processes that wish to transfer assets between accounts. This environment raises the need to tolerate the malicious behavior of processes that wish to sabotage the system.

In this work, we consider the shared memory model that was somewhat neglected in the Byzantine discussion. We believe that shared memory abstractions, implemented in distributed settings, allow for an intuitive formulation of the services offered by blockchains and similar decentralized tools. It is well-known that it is possible to implement reliable read-write shared memory registers via message passing even if a fraction of the servers are Byzantine [1, 21, 24, 19]. As a result, as long as the client processes using the service are not malicious, any fault-tolerant object that can be constructed using registers can also be implemented in the presence of Byzantine servers. However, it is not clear what can be done with such objects when they are used by Byzantine client processes. In this work, we study this question.

In Section 4 we define *Byzantine linearizability*, a correctness condition applicable to any shared memory object with a sequential specification. Byzantine linearizability addresses the usage of reliable shared memory abstractions by potentially Byzantine client processes. We then systematically study the feasibility of implementing various Byzantine linearizable shared memory objects from registers.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 18; pp. 18:1–18:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We observe that existing Byzantine fault-tolerant shared memory constructions [20, 22, 1] in fact implement Byzantine linearizable registers. Such registers are the starting point of our study. When trying to implement more complex objects (e.g., snapshots and asset transfer) using registers, constructions that work in the crash-failure model no longer work when Byzantine processes are involved, and new algorithms – or impossibility results – are needed.

As our first result, we prove in Section 5 that an asset transfer object used by Byzantine client processes does not have a wait-free implementation, even when its API is reduced to support only transfer operations (without reading processes' balances). Furthermore, it cannot be implemented without a majority of correct processes constantly taking steps. Asset transfer has wait-free implementations from both reliable broadcast [7] and snapshots [17] (which we adapt to a Byzantine version) and thus the same lower bound applies to reliable broadcast and snapshots as well.

In Section 6, we present a Byzantine linearizable reliable broadcast algorithm with resilience $f < \frac{n}{2}$, proving that, for this object, the resilience bound is tight. To do so, we define a sequential specification of a reliable broadcast object. Briefly, the object exposes broadcast and deliver operations and we require that deliver return messages previously broadcast. We show that a Byzantine linearizable implementation of such an object satisfies the classical (message-passing) definition [10]. Finally, in Section 7 we present a Byzantine linearizable snapshot with the same resilience. In contrast, previous constructions of Byzantine lattice agreement, which can be directly constructed from a snapshot [6], required $3f + 1$ processes to tolerate f failures.

All in all, we establish a tight bound on the resilience of emulations of three useful shared memory objects from registers. On the one hand, we show that it is impossible to obtain wait-free solutions as in the non-Byzantine model, and on the other hand, unlike previous snapshot and lattice agreement algorithms, our solutions do not require $n > 3f$. Taken jointly, our results yield the following theorem:

▶ **Theorem 1.** *In the Byzantine shared memory model, there exist n-process f-resilient Byzantine linearizable implementations of reliable broadcast, snapshot, and asset transfer objects from registers if and only if $f < \frac{n}{2}$.*

Although the construction of reliable registers in message passing systems requires $n > 3f$ servers, our improved resilience applies to client processes, which are normally less reliable than servers, particularly in the so-called *permissioned model* where servers are trusted and clients are ephemeral.

In summary, we make the following contributions:

- Formalizing Byzantine linearizability for any object with a sequential specification.
- Proving that some of the most useful building blocks in distributed computing, such as atomic snapshot and reliable broadcast, do not have f-resilient implementations from SWMR registers when $f \geq \frac{n}{2}$ processes are Byzantine.
- Presenting Byzantine linearizable implementations of a reliable broadcast object and a snapshot object with the optimal resilience.

2 Related Work

In [4] Aguilera et al. present a non-equivocating broadcast algorithm in shared memory. This broadcast primitive is weaker than reliable broadcast – it does not guarantee that all correct processes deliver the same messages, but rather that they do not deliver conflicting messages. A newer version of their work [5], developed concurrently and independently of our work[1], also implements reliable broadcast with $n \geq 2f+1$, which is very similar to our implementation. While the focus of their work is in the context of RDMA in the *M&M* (message–and–memory) model, our work focuses on the classical shared memory model, which can be emulated in classical message passing systems. While the algorithms are similar, we formulate reliable broadcast as a shared memory object, with designated API method signatures, which allows us to reason about the operation interval as needed for proving (Byzantine) linearizability and for using this object in constructions of other shared memory objects.

Given a reliable broadcast object, there are known implementations of lattice agreement [16, 26], which resembles a snapshot object. However, these constructions require $n = 3f+1$ processes. In our work, we present both Byzantine linearizable reliable broadcast and Byzantine snapshot, (from which Byzantine lattice agreement can be constructed [6]), with resilience $n = 2f+1$.

The asset transfer object we discuss in this paper was introduced by Guerraoui et al. [17, 15]. Their work provides a formalization of the cryptocurrency definition [23]. The highlight of their work is the observation that the asset transfer problem can be solved without consensus. It is enough to maintain a partial order of transactions in the systems, and in particular, every process can record its own transactions. They present a wait-free linearizable implementation of asset transfer in crash-failure shared memory, taking advantage of an atomic snapshot object. We show that we can use their solution, together with our Byzantine snapshot, to solve Byzantine linearizable asset transfer with $n = 2f+1$.

In addition, Guerraoui et al. present a Byzantine-tolerant solution in the message passing model. This algorithm utilizes reliable broadcast, where dependencies of transactions are explicitly broadcast along with the transactions. This solution does not translate to a Byzantine linearizable one, but rather to a sequentially consistent asset transfer object. In particular, reads can return old (superseded) values, and transfers may fail due to outdated balance reads.

Finally, recent work by Auvolat et al. [7] continues this line of work. They show that a FIFO order property between each pair of processes is sufficient in order to solve the asset transfer problem. This is because transfer operations can be executed once a process's balance becomes sufficient to perform a transaction and there is no need to wait for all causally preceding transactions. However, as a result, their algorithm is not sequentially consistent, or even causally consistent for that matter. For example, assume process i maintains an invariant that its balance is always at least 10, and performs a transfer with amount 5 after another process deposits 5 into its account, increasing its balance to 15. Using the protocol in [7], another process might observe i's balance as 5 if it sees i's outgoing transfer before the causally preceding deposit. Because our solution is Byzantine linearizable, such anomalies are prevented.

3 Model and Preliminaries

We study a distributed system in the shared memory model. Our system consists of a well-known static set $\Pi = \{1, \dots, n\}$ of asynchronous client processes. These processes have access to some shared memory objects. In the shared memory model, all communication between processes is done through the API exposed by the objects in the system: processes invoke operations that in turn, return some response to the process. In this work, we assume

[1] Their work [5] was in fact published shortly after the initial publication of our results [14].

a reliable shared memory. (Previous works have presented constructions of such reliable shared memory in the message passing model [1, 21, 24, 3, 19]). We further assume an adversary that may adaptively corrupt up to f processes in the course of a run. When the adversary corrupts a process, it is defined as *Byzantine* and may deviate arbitrarily from the protocol. As long as a process is not corrupted by the adversary, it is *correct*, follows the protocol, and takes infinitely many steps. In particular, it continues to invoke the object's API infinitely often. Later in the paper, we show that the latter assumption is necessary.

We enrich the model with a *public key infrastructure* (PKI). That is, every process is equipped with a public-private key pair used to sign data and verify signatures of other processes. We denote a value v signed by process i as $\langle v \rangle_i$.

Executions and Histories. We discuss algorithms emulating some object O from lower level objects (e.g., registers). An algorithm is organized as methods of O. A method execution is a sequence of *steps*, beginning with the method's invocation (invoke step), proceeding through steps that access lower level objects (e.g., register read/write), and ending with a return step. The invocation and response delineate the method's execution interval. In an *execution* σ of a Byzantine shared memory algorithm, each correct process invokes methods sequentially, where steps of different processes are interleaved. Byzantine processes take arbitrary steps regardless of the protocol. The *history* H of an execution σ is the sequence of high-level invocation and response events of the emulated object O in σ.

A *sub-history* of a history H is a sub-sequence of the events of H. A history H is *sequential* if it begins with an invocation and each invocation, except possibly the last, is immediately followed by a matching response. Operation *op* is pending in a history H if *op* is invoked in H but does not have a matching response event.

A history defines a partial order on operations: operation op_1 precedes op_2 in history H, denoted $op_1 \prec_H op_2$, if the response event of op_1 precedes the invocation event of op_2 in H. Two operations are concurrent if neither precedes the other.

Linearizability. A popular correctness condition for concurrent objects in the crash-fault model is linearizability [18], which is defined with respect to an object's sequential specification. A *linearization* of a concurrent history H of object o is a sequential history H' such that (1) after removing some pending operations from H and completing others by adding matching responses, it contains the same invocations and responses as H', (2) H' preserves the partial order $\prec_H$, and (3) H' satisfies o's sequential specification.

f-resilient. An algorithm is *f-resilient* if as long as at most f processes fail, every correct process eventually returns from each operation it invokes. A *wait-free* algorithm is a special case where $f = n - 1$.

Single Writer Multiple Readers Register. The basic building block in shared memory is a single writer multiple readers (SWMR) register that exposes *read* and *write* operations. Such registers are used to construct more complicated objects. The sequential specification of a SWMR register states that every read operation from register R returns the value last written to R. Note that if the writer is Byzantine, it can cause a correct reader to read arbitrary values.

Asset Transfer Object. In [17, 15], the asset transfer problem is formulated as a sequential object type, called *Asset Transfer Object*. The asset transfer object maintains a mapping from processes in the system to their balances[2]. Initially, the mapping contains the initial balances of all processes. The object exposes a *transfer* operation, *transfer(src,dst,amount)*, which can be invoked by process *src* (only). It withdraws *amount* from process *src*'s account and deposits it at process *dst*'s account provided that *src*'s balance was at least *amount*. It returns a boolean that states whether the transfer was successful (i.e., *src* had *amount* to spend). In addition, the object exposes a *read(i)* operation that returns the current balance of i.

4 Byzantine Linearizability

In this section we define Byzantine linearizability. Intuitively, we would like to tame the Byzantine behavior in a way that provides consistency to correct processes. We linearize the correct processes' operations and offer a degree of freedom to embed additional operations by Byzantine processes.

We denote by $H|_{correct}$ the projection of a history H to all correct processes. We say that a history H is Byzantine linearizable if $H|_{correct}$ can be augmented with operations of Byzantine processes such that the completed history is linearizable. That is, there is another history, with the same operations by correct processes as in H, and additional operations by another at most f processes. In particular, if there are no Byzantine failures then Byzantine linearizability is simply linearizability. Formally:

▶ **Definition 2** (Byzantine Linearizability). *A history H is Byzantine linearizable if there exists a history H' so that $H'|_{correct} = H|_{correct}$ and H' is linearizable.*

Similarly to linearizability, we say that an object is Byzantine linearizable if all of its executions are Byzantine Linearizable.

Next, we characterize objects for which Byzantine linearizability is meaningful. The most fundamental component in shared memory is read-write registers. Not surprisingly, such registers, whether they are single-writer or multi-writers ones are de facto Byzantine linearizable without any changes. This is because before every read from a Byzantine register, invoked by a correct process, one can add a corresponding Byzantine write.

In practice, multiple writers multiple readers (MWMR) registers are useless in a Byzantine environment as an adversary that controls the scheduler can prevent any communication between correct processes. SWMR registers, however, are still useful for constructing more meaningful objects. Nevertheless, the constructions used in the crash-failure model for linearizable objects do not preserve this property. For instance, if we allow Byzantine processes to run a classic atomic snapshot algorithm [2] using Byzantine linearizable SWMR registers, it will not result in a Byzantine linearizable snapshot object. The reason is that the algorithm relies on correct processes being able to perform "double-collect" meaning that at some point a correct process manages to read all registers twice without witnessing any changes. While this is true in the crash-failure model, in the Byzantine model this is not the case as the adversary can change some registers just before any correct read.

[2] The definition in [17] allows processes to own multiple accounts. For simplicity, we assume a single account per-process, as in [15].

Relationship to Other Correctness Conditions

Byzantine linearizability provides a simple and intuitive way to capture Byzantine behavior in the shared memory model. We now examine the relationship of Byzantine linearizability with previously suggested correctness conditions involving Byzantine processes.

PBFT [12, 11] presented a formalization of linearizability in the presence of Byzantine-faulty clients in message passing systems. Their notion of linearizability is formulated in the form of I/O automata. Their specification is in the same spirit as ours, but our formulation is closer to the original notion of linearizability in shared memory.

Some works have defined linearization conditions for specific objects. This includes conditions for SWMR registers [22], a distributed ledger [13], and asset transfer [7]. Our condition coincides with these definitions for the specific objects and thus generalizes all of them. Liskov and Rodrigues [20] presented a correctness condition that has additional restrictions. Their correctness notion relies on the idea that Byzantine processes are eventually detected and removed from the system and focuses on converging to correct system behavior after their departure. While this model is a good fit when the threat model is software bugs or malicious intrusions, it is less appropriate for settings like cryptocurrencies, where Byzantine behavior cannot be expected to eventually stop.

5 Lower Bound on Resilience

In shared memory, one typically aims for wait-free objects, which tolerate any number of process failures. Indeed, many useful objects have wait-free implementations from SWMR registers in the non-Byzantine case. This includes reliable broadcast, snapshots, and as recently shown, also asset transfer. We now show that in the Byzantine case, wait-free implementations of these objects are impossible. Moreover, a majority of correct processes is required.

▶ **Theorem 3.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist a Byzantine linearizable implementation of asset transfer that supports only transfer operations in a system with $n \leq 2f$ processes, f of which can be Byzantine, using only SWMR registers.*

Note that to prove this impossibility, it does not suffice to introduce bogus actions by Byzantine processes, because the notion of Byzantine linearizability allows us to ignore these actions. Rather, to derive the contradiction, we create runs where the bogus behavior of the Byzantine processes leads to incorrect behavior of the correct processes.

Proof. Assume by contradiction that there is such an algorithm. Let us look at a system with $n = 2f$ correct processes. Partition Π as follows: $\Pi = A \cup B \cup \{p_1, p_2\}$, where $|A| = f - 1$, $|B| = f - 1$, $A \cap B = \emptyset$, and $p_1, p_2 \notin A \cup B$. By assumption, $|A| > 1$. Let z be a process in A. Also, by assumption $|B| \geq 2$. Let q_1, q_2 be processes in B. The initial balance of all processes but z is 0, and the initial balance of z is 1. We construct four executions as shown in Figure 1.

Let σ_1 be an execution where, only processes in $A \cup \{p_1\}$ take steps. First, z performs *transfer(z,p_1,1)*. Since up to f processes may be faulty, the operation completes, and by the object's sequential specification, it is successful (returns true). Then, p_1 performs *transfer(p_1,q_1,1)*. By f-resilience and linearizability, this operation also completes successfully. Note that in σ_1 no process is actually faulty, but because of f-resilience, progress is achieved when f processes are silent.

Similarly, let σ_2 be an execution where the processes in $A \cup \{p_2\}$ are correct, and z performs *transfer(z,p_2,1)*, followed by p_2 performing *transfer(p_2,q_2,1)*.

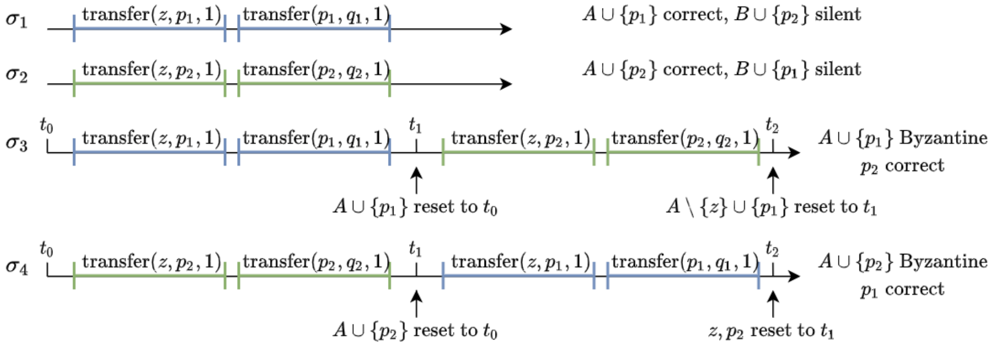


Figure 1 An asset transfer object does not have an f-resilient implementation for $n \leq 2f$.

We now construct σ_3, where all processes in $A\cup\{p_1\}$ are Byzantine. We first run σ_1. Call the time when it ends t_1. At this point, all processes in $A\cup\{p_1\}$ restore their registers to their initial states. Note that no other processes took steps during σ_1, hence the entire shared memory is now in its initial state. Then, we execute σ_2. Because we have reset the memory to its initial state, the operations execute the same way. When σ_2 completes, processes in $A\backslash\{z\}\cup\{p_1\}$ restore their registers to their state at time t_1. At this point, the state of z and p_2 is the same as it was at the end of σ_2, the state of processes in $A\setminus\{z\}\cup\{p_1\}$ is the same as it was at the end of σ_1, and processes in B are all in their initial states.

We construct σ_4 where all processes in $A\cup\{p_2\}$ are Byzantine by executing σ_2, having all processes in $A\cup\{p_2\}$ reset their memory, executing σ_1, and then having z and p_2 restore their registers to their state at the end of σ_2. At this point, the state of z and p_2 is the same as it was at the end of σ_2, the state of processes in $A\setminus\{z\}\cup\{p_1\}$ is the same as it was at the end of σ_1, and processes in B are all in their initial states.

We observe that for processes in B, the configurations at the end of σ_3 and σ_4 are indistinguishable as they did not take any steps and the global memory is the same. By f-resilience, in both cases q_1 and q_2, together with processes in B and one of $\{p_1,p_2\}$ should be able to make progress at the end of each of these runs. We extend the runs by having q_1 and q_2 invoke transfers of amount 1 to each other. In both runs processes in $B\cup\{p_1,p_2\}$ help them make progress. In σ_3, p_1 behaves as if it is a correct process and its local state is the same as it is at the end of σ_1, and in σ_4 p_2 behaves as if it is a correct process and its local state is the same as it is at the end of σ_2. Thus, σ_3 and σ_4 are indistinguishable to all correct processes, and as a result q_1 and q_2 act the same in both runs. However, from safety exactly one of their transfers should succeed. In σ_3, p_2 is correct and *transfer*(p_2,q_2,*1*) succeeds, allowing q_2 to transfer 1 and disallowing the transfer from q_1, whereas σ_4 the opposite is true. This is a contradiction. ◀

Guerraoui et al. [17] use an atomic snapshot to implement an asset transfer object in the crash-fault shared memory model. In addition, they handle Byzantine processes in the message passing model by taking advantage of reliable broadcast. In Appendix A we show that their atomic snapshot-based asset transfer can be easily adapted to the Byzantine settings by using a Byzantine linearizable snapshot, resulting in a Byzantine linearizable asset transfer. Their reliable broadcast-based algorithm, on the other hand, is not linearizable and therefore not Byzantine linearizable even when using Byzantine linearizable reliable broadcast. Nonetheless, Auvolat et al. [7] have used reliable broadcast to construct an asset transfer object where transfer operations are linearizable (although reads are not).

We note that our lower bound holds for an asset transfer object without read operations. In Algorithm 4 in Appendix A we construct an asset transfer object given a Byzantine linearizable snapshot (proofs appear in the full version [14]). The above discussion and the construction in Algorithm 4 lead us to the following corollary:

▶ **Corollary 4.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist an f-resilient Byzantine linearizable implementation of an atomic snapshot or reliable broadcast in a system with $f \geq \frac{n}{2}$ Byzantine processes using only SWMR registers.*

Furthermore, we prove in the following lemma that in order to provide f-resilience it is required that at least a majority of correct processes take steps infinitely often, justifying our model definition.

▶ **Lemma 5.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist an f-resilient Byzantine linearizable implementation of asset transfer in a system with $n \geq 2f+1$ processes, f of which can be Byzantine, using only SWMR registers if less than $f+1$ correct processes take steps infinitely often.*

Proof. Assume by way of contradiction that there exists an f-resilient Byzantine linearizable implementation of asset transfer in a system with $n \geq 2f + 1$ processes where there are at most f correct processes that take steps infinitely often. Denote these f correct processes by the set A. Thus, there is a point t in any execution such that from time t, only processes in A and Byzantine processes take any steps. Starting t, the implementation is equivalent to one in a system with $n = 2f$, f of them may be Byzantine. This is a contradiction to Theorem 3. ◀

6 Byzantine Linearizable Reliable Broadcast

With the acknowledgment that not all is possible, we seek to find Byzantine linearizable objects that are useful even without a wait-free implementation. One of the practical objects is a reliable broadcast object. We already proved in the previous section that it does not have an f-resilient Byzantine linearizable implementation, for any $f \geq max\{3, \frac{n}{2}\}$. In this section we provide an implementation that tolerates $f < \frac{n}{2}$ faults.

6.1 Reliable Broadcast Object

The reliable broadcast primitive exposes two operations *broadcast(ts,m)* returning void and *deliver(j,ts)* returning m. When $deliver_j(i, ts)$ returns m we say that process j delivers m from process i in timestamp ts. The broadcast operation allows processes to spread a message m in the system, along with some timestamp ts. The use of timestamps allows processes to broadcast multiple messages.

Its classical definition, given for message passing systems [10], requires the following properties:

- Validity: If a correct process i broadcasts (ts, m) then all correct processes eventually deliver m from process i in timestamp ts.
- Agreement: If a correct process delivers m from process i in timestamp ts, then all correct processes eventually deliver m from process i in timestamp ts.
- Integrity: No process delivers two different messages for the same (ts, j) and if j is correct delivers only messages j previously broadcast.

In the shared memory model, the deliver operation for some process j and timestamp ts returns the message with timestamp ts previously broadcast by j, if exists. We define the sequential specification of reliable broadcast as follows:

▶ **Definition 6.** *A reliable broadcast object exposes two operations* broadcast(ts,m) *returning void and* deliver(j,ts) *returning m. A call to* deliver(j,ts) *returns the value m of the first* broadcast(ts,m) *invoked by process j before the deliver operation. If j did not invoke* broadcast *before the deliver, then it returns* $\perp$.

Note that as the definition above refers to sequential histories, the first broadcast operation (if such exists) is well-defined. Further, whereas in message passing systems reliable broadcast works in a push fashion, where the receipt of a message triggers action at its destination, in the shared memory model processes need to actively pull information from the registers. A process pulls from another process j using the *deliver(j,ts)* operation and returns with a value $m \neq \perp$. If all messages are eventually pulled, the reliable broadcast properties are achieved, as proven in the following lemma.

▶ **Lemma 7.** *A Byzantine linearization of a reliable broadcast object satisfies the three properties of reliable broadcast.*

Proof. If a correct process broadcasts m, and all messages are subsequently pulled then according to Definition 6 all correct processes deliver m, providing validity. For agreement, if a correct process invokes *deliver(j,ts)* that returns m and all messages are later pulled by all correct processes, it follows that all correct processes also invoke *deliver(j,ts)* and eventually return $m' \neq \perp$. Since *deliver(j,ts)* returns the value v of the first *broadcast(ts,v)* invoked by process j before it is called, and there is only one first broadcast, and we get that $m = m'$. Lastly, if *deliver(j,ts)* returns m, by the specification, j previously invoked *broadcast(ts,m)*. ◀

6.2 Reliable Broadcast Algorithm

In our implementation (given in Algorithm 1), each process has 4 SWMR registers: send, echo, ready, and deliver, to which we refer as *stages* of the broadcast. We follow concepts from Bracha's implementation in the message passing model [9] but leverage the shared memory to improve its resilience from $3f + 1$ to $2f + 1$. The basic idea is that a process that wishes to broadcast value v writes it in its send register (line 4) and returns only when it reaches the deliver stage. I.e., v appears in the deliver register of at least one correct process. Throughout the run, processes infinitely often call a *refresh* function whose role is to help the progress of the system. When refreshing, processes read all registers and help promote broadcast values through the 4 stages. For a value to be delivered, it has to have been read and signed by $f + 1$ processes at the ready stage. Because each broadcast message is copied to 4 registers of each process, the space complexity is $4n$ per message. Whether this complexity can be improved remains as an open question.

In the refresh function, executed for all processes, at first a process reads the last value written to a send register (line 16). If the value is a signed pair of a message and a timestamp, refresh then copies it to the process's echo register in line 18. In the echo register, the value remains as evidence, preventing conflicting values (sent by Byzantine processes) from being delivered. That is, before promoting a value to the ready or deliver stage, a correct process i performs a "double-collect" of the echo registers (in lines 19,21). Namely, after collecting $f + 1$ signatures on a value in ready registers, meaning that it was previously written in the echo of at least one correct process, i re-reads all echo registers to verify that there does not exist a conflicting value (with the same timestamp and sender). Using this method, concurrent deliver operations “see” each other, and delivery of conflicting values broadcast by a Byzantine process is prevented. Before delivering a value, a process writes it to its deliver register with $f+1$ signatures (line 22). Once one correct process delivers a value, the following deliver calls can witness the $f+1$ signatures and copy this value directly from its deliver register (line 11).

Algorithm 1 Shared Memory Bracha: code for process i.

```
    shared SWMR registers: $send_i, echo_i, ready_i, deliver_i$

 1: procedure CONFLICTING-ECHO($\langle ts, v\rangle_j$)
 2:     return $\exists w \neq v, k \in \Pi$ such that $\langle ts, w\rangle_j \in echo_k$

 3: procedure BROADCAST(ts,val)
 4:     $send_i \leftarrow \langle ts, val\rangle_i$
 5:     repeat
 6:         $m \leftarrow$ deliver(i,ts)
 7:     until $m \neq \bot$                                   ▷ message is deliverable

 8: procedure DELIVER(j,ts)
 9:     refresh()
10:     if $\exists k \in \Pi$ and $v$ s.t. $\langle\langle ts, v\rangle_j, \sigma\rangle \in deliver_k$ where $\sigma$ is a set of $f+1$ signatures on
    $\langle ready, \langle ts, v\rangle_j\rangle$ then
11:         $deliver_i \leftarrow deliver_i \cup \{\langle\langle ts, v\rangle_j, \sigma\rangle\}$
12:         return $v$
13:     return $\bot$

14: procedure REFRESH
15:     for $j \in [n]$ do
16:         $m \leftarrow send_j$
17:         if $\nexists ts$, $val$ s.t. $m = \langle ts, val\rangle_j$ then continue          ▷ $m$ is not a signed pair
18:         $echo_i \leftarrow echo_i \cup \{m\}$
19:         if $\neg$conflicting-echo($m$) then
20:             $ready_i \leftarrow ready_i \cup \{\langle ready, m\rangle_i\}$
21:         if $\exists S \subseteq \Pi$ s.t. $|S| \geq f+1$, $\forall j \in S, \langle ready, m\rangle_j \in ready_j$ and $\neg$conflicting-echo($m$)
    then
22:             $deliver_i \leftarrow deliver_i \cup \{\langle m, \sigma = \{\langle ready, m\rangle_j | j \in S\}\rangle\}$    ▷ $\sigma$ is the set of $f+1$
    signatures
```

We make two assumptions on the correct usage of our algorithm. The first is inherently required as shown in Lemma 5:

▶ **Assumption 1.** *All correct processes infinitely often invoke methods of the reliable broadcast API.*

The second is a straight forward validity assumption:

▶ **Assumption 2.** *Correct processes do not invoke* broadcast(ts,val) *twice with the same ts.*

In the full version [14] we prove the correctness of the reliable broadcast algorithm and conclude the following theorem:

▶ **Theorem 8.** *Algorithm 1 implements an f-resilient Byzantine linearizable reliable broadcast object for any $f < \frac{n}{2}$.*

7 Byzantine Linearizable Snapshot

In this section, we utilize a reliable broadcast primitive to construct a Byzantine snapshot object with resilience $n > 2f$.

7.1 Snapshot Object

A snapshot [2] is represented as an array of n shared single-writer variables that can be accessed with two operations: *update(v)*, called by process i, updates the i^{th} entry in the array and *snapshot* returns an array. The sequential specification of an atomic snapshot is as follows: the i^{th} entry of the array returned by a *snapshot* invocation contains the value v last updated by an *update(v)* invoked by process i, or its variable's initial value if no update was invoked.

Following Lemma 5, we again must require that correct processes perform operations infinitely often. For simplicity, we require that they invoke infinitely many snapshot operations; if processes invoke either snapshots or updates, we can have each update perform a snapshot and ignore its result.

▶ **Assumption 3.** *All correct processes invoke snapshot operations infinitely often.*

7.2 Snapshot Algorithm

Our pseudo-code is presented in Algorithms 2 and 3. During the algorithm, we compare snapshots using the (partial) coordinate-wise order. That is, let s_1 and s_2 be two n-arrays. We say that $s_2 > s_1$ if $\forall i \in [n]$, $s_2[i].ts > s_1[i].ts$.

Recall that all processes invoke snapshot operations infinitely often. In each snapshot instance, correct processes start by collecting values from all registers and broadcasting their collected arrays in "start" messages (message with timestamp 0). Then, they repeatedly send the identities of processes from which they delivered start messages until there exists a round such that the same set of senders is received from $f + 1$ processes in that round. Once this occurs, it means that the $f + 1$ processes see the exact same start messages and the snapshot is formed as the supremum of the collects in their start messages.

We achieve optimal resilience by waiting for only $f + 1$ processes to send the same set. Although there is not necessarily a correct process in the intersection of two sets of size $f+1$, we leverage the fact that reliable broadcast prevents equivocation to ensure that nevertheless, there is a common *message* in the intersection, so two snapshots obtained in the same round are necessarily identical. Moreover, once one process obtains a snapshot s, any snapshot seen in a later round exceeds s.

Each process i collects values from all processes' registers in a shared variable $collect_i$. When starting a snapshot operation, each process runs update-collect, where it updates its collect array (line 8) and saves it in a local variable c (line 9). When it does so, it updates the i^{th} entry to be the highest-timestamped value it observes in the i^{th} entries of all processes' collect arrays (lines 16 – 18). Then, it initiates the snapshot-aux procedure with a new

Algorithm 2 Byzantine Snapshot: code for process i.

```
    shared SWMR registers: $\forall j \in [n]$ $collected_i[j] \in \{\bot\} \cup \{\mathbb{N} \times Vals\}$ with selectors $ts$ and val, initially $\bot$
    $\forall k \in \mathbb{N}$, $savesnap_i[k] \in \{\bot\} \cup \{$array of $n$ $Vals \times$ set of messages$\}$ with selectors $snap$ and proof, initially $\bot$
    local variables: $ts_i \in \mathbb{N}$, initially 0
    $\forall j \in [n]$, $rts_i[j] \in \mathbb{N}$, initially 0
    $r, auxnum \in \mathbb{N}$, initially 0
    $p \in [n]$, initially 1
    $\forall j \in [n], k \in \mathbb{N}$, $seen_i[j][k]$,$senders_i \in \mathcal{P}(\Pi)$, initially $\emptyset$
    $\sigma \leftarrow \emptyset$ set of messages

 1: procedure UPDATE($v$)
 2:     for $j \in [n]$ do                                  ▷ collect current memory state
 3:         update-collect($collected_j$)
 4:     $ts_i \leftarrow ts_i + 1$
 5:     $collected_i[i] \leftarrow \langle ts_i, v \rangle_i$            ▷ update local component of collected

 6: procedure SNAPSHOT
 7:     for $j \in [n]$ do                                  ▷ collect current memory state
 8:         update-collect($collected_j$)
 9:     $c \leftarrow collected_i$
10:     repeat
11:         $auxnum \leftarrow auxnum + 1$
12:         $snap \leftarrow$ snapshot-aux($auxnum$)
13:     until $snap \geq c$                      ▷ snapshot is newer than the collected state
14:     return $snap$

15: procedure UPDATE-COLLECT(c)
16:     for $k \in [n]$ do
17:         if $c[k].ts > collected_i[k].ts$ and $c[k]$ is signed by $k$ then
18:             $collected_i[k] \leftarrow c[k]$
```

auxnum tag. Snapshot-aux returns a snapshot, but not necessarily a "fresh" one that reflects all updates that occurred before *snapshot* was invoked. Therefore, snapshot-aux is repeatedly called until it collects a snapshot s such that $s \geq c$, according to the snapshots partial order (lines 10 – 13).

By Assumption 3 and since the *auxnum* variable at each correct process is increased by 1 every time snapshot-aux is called, all correct processes participate in all instances of snapshot-aux. When a correct process invokes a snapshot-aux procedure with auxnum, it first initiates a new reliable broadcast instance at line 28, dedicated to this instance of snapshot-aux. Note that although processes invoke one snapshot-aux at a time, they may engage in multiple reliable broadcast instances simultaneously. That is, they continue to partake in previous reliable broadcast instances after starting a new one. As another preliminary step of snapshot-aux, each correct process once again updates its collect array using the update-collect procedure (lines 30– 31) and broadcasts it to all processes at line 33.

Algorithm 3 Byzantine Snapshot auxiliary procedures: code for process i.

```
procedure MINIMUM-SAVED(auxnum)
    $S \leftarrow \{s | \exists j \in [n], s = savesnap_j[auxnum].snap$ and $savesnap_j[auxnum].proof$ is a valid proof of $s\}$
    if $S = \emptyset$ then
        return $\perp$
    $res \leftarrow$ infimum($S$)                ▷ returns the minimum value in each index
    $savesnap_i[auxnum] \leftarrow \langle res, \bigcup_{j \in [n]} savesnap_j[auxnum].proof \rangle$
    update-collect($res$)
    return $res$

procedure SNAPSHOT-AUX(auxnum)
    initiate new reliable broadcast instance
    $\sigma \leftarrow \emptyset$
    for $j \in [n]$ do                ▷ collect current memory state
        update-collect($collected_j$)
    $senders_i \leftarrow \{i\}$                ▷ start message contains collect
    broadcast(0,$\langle collect_i \rangle_i$)
    while true do
        $cached \leftarrow$ minimum-saved($auxnum$)                ▷ check if there is a saved snapshot
        if $cached \neq \perp$ then return $cached$
        $p \leftarrow (p + 1)$ mod $n + 1$                ▷ deliver messages in round robin
        $m \leftarrow$ deliver($p, rts_i[p]$)                ▷ deliver next message from $p$
        if $m = \perp$ then  continue
        if $rts_i[p]$= 0 and $m$ contains a signed collect array $c$ then
                                                ▷ start message (round 0)
            $\sigma \leftarrow \sigma \cup \{m\}$
            update-collect($c$)
            $senders_i \leftarrow senders_i \cup \{j\}$
        else if $m$ contains a signed set of processes, $jsenders$ then
                                                ▷ round $r$ message for $r > 0$
            if $jsenders \not\subseteq senders_i$ then
                continue                ▷ cannot process message, its dependencies are missing
            $\sigma \leftarrow \sigma \cup \{m\}$
            $seen_i[j][rts_i[p]] \leftarrow jsenders \cup seen_i[j][rts_i[p] - 1]$
        $rts_i[p] \leftarrow rts_i[p] + 1$

        if received $f + 1$ round-$r$ messages for the first time then
            $r \leftarrow r + 1$
            broadcast($r$, $\langle senders_i \rangle_i$)

        if $\exists s$ s.t. $|\{j|\ seen_i[j][s] = senders_i\}| = f + 1$ then                ▷ stability condition
            $r \leftarrow 0$
            $senders_i \leftarrow \emptyset$
            $\forall j \in [n], k \in \mathbb{N},\ seen_i[j][k] \leftarrow \emptyset$
            $cached \leftarrow$ minimum-saved($auxnum$)                ▷ re-check for saved snapshot
            if $cached \neq \perp$ then return $cached$
            $savesnap_i[auxnum] \leftarrow \langle collect_i, \sigma \rangle$
                                ▷ $\sigma$ contains all received messages in this snapshot-aux instance
            return $collect_i$
```

During the execution, a correct processes delivers messages from all other processes in a round robin fashion. The local variable p represents the process from which it currently delivers. In addition, $rts[p]$ maintains the next timestamp to be delivered from p (lines 38, 49, 37). Note that if the delivered message at some point is $\perp$, $rts[p]$ is not increased, so all of p's messages are delivered in order (line 39).

Snapshot-aux proceeds in rounds, which are reflected in the timestamps of the messages broadcast during its execution. Each correct process starts snapshot-aux at round 0, where it broadcasts its collected array; we refer to this as its start message. It then continues to round $r + 1$ once it has delivered $f + 1$ round r messages (line 51). Each process maintains a local set *senders* that contains the processes from which it received start messages (line 43). In every round (from 1 onward) processes send the set of processes from which they received start messages (line 52).

Process i maintains a local map $seen[j][r]$ that maps a process j and a round r to the set of processes that j reported to have received start messages from in rounds 1–r (line 48), but only if i has received start messages from all the reported processes (line 45). By doing so, we ensure that if for some correct process i and a round r $seen_i[j][r]$ contains a process l, l is also in $senders_i$. If this condition is not satisfied, the delivered counter for j ($rts[j]$) is not increased and this message will be repeatedly delivered until the condition is satisfied.

Once there is a process i such that there exists a round s and there is a set S of $f + 1$ processes j for which $seen_i[j][s]$ is equal to $senders_i$, we say that the *stability condition* at line 53 is satisfied for S. At that time, i and f more processes agree on the collected arrays sent at round 0 by processes in $senders_i$, and $collect_i$ holds the supremum of those collected arrays. This is because whenever it received a start message, it updated its collect so that currently $collect_i$ reflects all collects sent by processes in $senders_i$. Thus, i can return its current collect as the snapshot-aux result. Since reliable broadcast prevents Byzantine processes from equivocating, there are f more processes that broadcast the same *senders* set at that round, and any future round will "see" this set. As we later show, after at most $n + 1$ rounds, the stability condition holds and hence the size of *seen* is $O(n^3)$. Together with the collected arrays, the total space complexity is cubic in n.

To ensure liveness in case some correct processes complete a snapshot-aux instance before all do, we add a helping mechanism. Whenever a correct process successfully completes snapshot-aux, it stores its result in a savesnap map, with the auxnum as the key (either at line 24 or at line 59). This way, once one correct process returns from snapshot-aux, others can read its result at line 35 and return as well. To prevent Byzantine processes from storing invalid snapshots, each entry in the savesnap map is a tuple of the returned array and a proof of the array's validity. The proof is the set of messages received by the process that stores its array in the current instance of snapshot-aux. Using these messages, correct processes can verify the legitimacy of the stored array. If a correct process reads from savesnap a tuple with an invalid proof, it simply ignores it.

7.3 Correctness

We outline the key correctness arguments highlighting the main lemmas. Formal proofs of these lemmas appear in the full version [14]. To prove our algorithm is Byzantine linearizable, we first show that all returned snapshots are totally ordered (by coordinate-wise order):

▶ **Lemma 9.** *If two snapshot operations invoked by correct processes return s_i and s_j, then $s_j \geq s_i$ or $s_j < s_i$.*

Based on this order, we define a linearization. Then, we show that our linearization preserves real-time order, and it respects the sequential specification. We construct the linearization E as follows: First, we linearize all snapshot operations of correct processes in the order of their return values. Then, we linearize every update operation by a correct process immediately before the first snapshot operation that "sees" it. We say that a snapshot returning s *sees* an update by process j that has timestamp ts if $s[j].ts \geq ts$. If multiple updates are linearized to the same point (before the same snapshot), we order them by their start times. Finally, we add updates by Byzantine processes as follows: We add *update(v)* by a Byzantine process j if there is a linearized snapshot that returns s and $s[j].val = v$. We add the update immediately before any snapshot that sees it.

We next prove that the linearization respects the sequential specification.

▶ **Lemma 10.** *The i^{th} entry of the array returned by a* snapshot *invocation contains the value v last updated by an* update(v) *invoked by process i in E, or its variable's initial value if no update was invoked.*

Because an update is linearized immediately before some snapshot sees it and snapshots are monotonically increasing, all following snapshots see the update as well. Next, we prove in the two following lemmas that E preserves the real-time order.

▶ **Lemma 11.** *If a snapshot operation invoked by a correct process i with return value s_i precedes a snapshot operation invoked by a correct process j with return value s_j, then $s_i \leq s_j$.*

▶ **Lemma 12.** *Let s be the return value of a snapshot operation $snap_i$ invoked by a correct process i. Let $update_j(v)$ be an update operation invoked by a correct process j that writes $\langle ts, v\rangle$ and completes before $snap_i$ starts. Then, $s[j].ts \geq ts$.*

It follows from Lemma 12 and the definition of E, that if an update precedes a snapshot it is linearized before it, and from Lemma 11 that if a snapshot precedes a snapshot it is also linearized before it. The following lemma ensures that if an update precedes another update it is linearized before it. That is, if a snapshot operation sees the second update, it sees the first one.

▶ **Lemma 13.** *If update1 by process i precedes update2 by process j and a snapshot operation snap by a correct process sees update2, then snap sees update1 as well.*

Finally, the next lemma proves the liveness of our algorithm.

▶ **Lemma 14.** *(Liveness) Every correct process that invokes some operation eventually returns.*

We conclude the following theorem:

▶ **Theorem 15.** *Algorithm 2 implements an f-resilient Byzantine linearizable snapshot object for any $f < \frac{n}{2}$.*

Proof. Lemma 9 shows that there is a total order on snapshot operations. Using this order, we have defined a linearization E that satisfies the sequential specification (Lemma 10). We then proved that E also preserves real-time order (Lemmas 11 – 13). Thus, Algorithm 2 is Byzantine linearizable. In addition, Lemma 14 proves that Algorithm 2 is f-resilient. ◀

8 Conclusions

We have studied shared memory constructions in the presence of Byzantine processes. To this end, we have defined Byzantine linearizability, a correctness condition suitable for shared memory algorithms that can tolerate Byzantine behavior. We then used this notion to present both upper and lower bounds on some of the most fundamental components in distributed computing.

We proved that atomic snapshot, reliable broadcast, and asset transfer are all problems that do not have f-resilient emulations from registers when $n \leq 2f$. On the other hand, we have presented an algorithm for Byzantine linearizable reliable broadcast with resilience $n > 2f$. We then used it to implement a Byzantine snapshot with the same resilience. Among other applications, this Byzantine snapshot can be utilized to provide a Byzantine linearizable asset transfer. Thus, we proved a tight bound on the resilience of emulations of asset transfer, snapshot, and reliable broadcast.

Our paper deals with feasibility results and does not focus on complexity measures. In particular, we assume unbounded storage in our constructions. We leave the subject of efficiency as an open question for future work.

References

1 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

3 Yehuda Afek, David S Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM (JACM)*, 42(6):1231–1274, 1995.

4 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

5 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement, 2021. arXiv:1905.12143.

6 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement. In *International Workshop on Distributed Algorithms*, pages 35–53. Springer, 1992.

7 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bulletin of EATCS*, 3(132), 2020.

8 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

10 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

11 Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

12 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

13 Vicent Cholvi, Antonio Fernandez Anta, Chryssis Georgiou, Nicolas Nicolaou, and Michel Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 77–84. IEEE, 2020.

14 Shir Cohen and Idit Keidar. Tame the wild with byzantine linearizability: Reliable broadcast, snapshots, and asset transfer. *arXiv preprint*, 2021. arXiv:2102.10597.

15 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

16 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 674–683. IEEE, 2020.

17 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

18 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

19 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, 1998.

20 Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *International Symposium on Distributed Computing*, pages 487–489. Springer, 2005.

21 Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pages 311–325. Springer, 2002.

22 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60(4):677–694, 2017.

23 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2009.

24 Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, 2003.

25 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

26 Xiong Zheng and Vijay K. Garg. Byzantine lattice agreement in asynchronous systems. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14–16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.4.

A Byzantine Asset Transfer

In this section we adapt the asset transfer implementation from snapshots given in [17] to a Byzantine asset transfer. The algorithm is very simple. It is based on a shared snapshot array S, with a cell for each client process i, representing i's outgoing transactions. An additional immutable array holds all processes' initial balances. A process i's balance is computed by taking a snapshot of S and applying all of i's valid incoming and outgoing transfers to i's initial balance. A transfer invoked by process i checks if i's balance is sufficient, and if so, appends the transfer details (source, destination, and amount) to i's cell. Similarly to the use of dependencies in the (message-passing broadcast-based) asset transfer algorithm of [17], we also track the history of every transaction. To this end, we append to the process's cell also the snapshot taken to compute the balance for each transaction.

Algorithm 4 Byzantine Asset Transfer: code for process i.

```
shared Byzantine snapshot: $S$
initial– immutable array of initial balances
local variables: $txns_i$ – sets of outgoing transaction, initially $\{\}$
$ts_i \in \mathbb{N}$, initially 0
$snap$ – array of sets of transactions, initially array of empty sets    ▷ the last snapshot taken

struct $txn$ contains:
    timestamp ts,
    source src,
    destination dst,
    amount amount

procedure BALANCE(j,snap)
    $incoming \leftarrow 0$
    $outgoing \leftarrow 0$
    for $l \in [n]$ do
        for $k \in snap[l]$ do
            if $snap[l][k].dst = j$ and valid$(snap[l][k])$ then
                $incoming \leftarrow incoming + snap[l][k].amount$
    for $k \in snap[j]$ do
        if valid$(snap[j][k])$ then
            $outgoing \leftarrow outgoing + snap[j][k].amount$
    return $initial(j) + incoming - outgoing$

procedure TRANSFER(src,dst,amount)
    $ts_i \leftarrow ts_i + 1$
    $snap \leftarrow S.snapshot()$
    if $balance(src, snap) < amount$ then
        return false
    $txns_i \leftarrow txns_i.append(\langle ts_i, src, dst, amount, snap\rangle_i)$
    $S.update(txns_i)$
    return true

procedure READ(j)
    $snap \leftarrow S.snapshot()$
    return $balance(j, snap)$
```

Wake up and Join Me! an Energy-Efficient Algorithm for Maximal Matching in Radio Networks

Varsha Dani
Department of Computer Science, Rochester Institute of Technology, NY, USA

Aayush Gupta
Department of Computer Science, University of New Mexico, Albuquerque, NM, USA

Thomas P. Hayes
Dept. of Computer Science, University of New Mexico, Albuquerque, NM, USA

Seth Pettie
Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA

Abstract

We consider networks of small, autonomous devices that communicate with each other wirelessly. Minimizing energy usage is an important consideration in designing algorithms for such networks, as battery life is a crucial and limited resource. Working in a model where both sending and listening for messages deplete energy, we consider the problem of finding a maximal matching of the nodes in a radio network of arbitrary and unknown topology.

We present a distributed randomized algorithm that produces, with high probability, a maximal matching. The maximum energy cost per node is $O(\log^2 n)$, and the time complexity is $O(\Delta \log n)$. Here n is any upper bound on the number of nodes, and Δ is any upper bound on the maximum degree; n and Δ are parameters of our algorithm that we assume are known *a priori* to all the processors. We note that there exist families of graphs for which our bounds on energy cost and time complexity are simultaneously optimal up to polylog factors, so any significant improvement would need additional assumptions about the network topology.

We also consider the related problem of assigning, for each node in the network, a neighbor to back up its data in case of eventual node failure. Here, a key goal is to minimize the maximum *load*, defined as the number of nodes assigned to a single node. We present an efficient decentralized low-energy algorithm that finds a neighbor assignment whose maximum load is at most a polylog(n) factor bigger that the optimum.

2012 ACM Subject Classification Networks

Keywords and phrases Distributed Algorithms, Energy-Aware Computation, Radio Networks, Maximal Matching, Sensor Networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.19

Funding *Seth Pettie*: Funded by NSF grant CCF-1815316.

1 Introduction

For networks of small computers, energy management and conservation is often a major concern. When these networks communicate wirelessly, usage of the radio transceiver to send or listen for messages is often one of the dominant causes of energy usage. Moreover, this has tended to be increasingly true as the devices have gotten smaller; see, for example, [13, 3, 10]. Motivated by these considerations, Chang et al. [4] introduced a theoretical model of distributed computation in which each send or listen operation costs one unit of energy, but local computation is free. Over a sequence of discrete timesteps, nodes choose whether

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 19; pp. 19:1–19:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to sleep, listen, or send a message of $O(\log n)$ bits. A listening node successfully receives a message only when exactly one of its neighbors has chosen to send in that timestep; otherwise it receives no input.

It is not uncommon for research on sensor networks to make assumptions about the topology of the network, such as assuming the network is defined by a unit disk graph, or that each node is aware of its location using GPS. However, we will be interested in the more general setting where we make almost no assumptions about the network topology. We will assume that communication takes place via radio broadcasts, and that there is an arbitrary and unknown undirected graph G whose edges indicate which pairs of nodes are capable of hearing each other's broadcasts. We will, however, assume that each node is initialized with shared parameters n and Δ, which are upper bounds on, respectively, the total number of nodes, and the maximum degree of any node. By designing algorithms to operate without pre-conditions on, or foreknowledge of, the network topology, we potentially broaden the possible applications of our algorithms, and, by extension, of sensor networks. For instance, we can imagine a network of small sensors scattered rather haphazardly from an airplane passing over hazardous terrain; the sensors that survive their landing are unlikely to be placed predictably or uniformly.

In this model, [4] presented a polylog-energy, polynomial-time algorithm for the problem of one-to-all broadcast. A later paper, by Chang, Dani, Hayes, and Pettie [5], gave a sub-polynomial ($n^{o(1)}$) energy, polynomial-time algorithm for the related problem of breadth-first search.

In the present work, we will be concerned with another fundamental problem of graph theory, namely to find large sets of pairwise disjoint edges, or *matchings*. The problem of finding large matchings has been thoroughly studied in a wide variety of computational models dating back more than a century, to König [11]. For a fairly comprehensive review of past results, we recommend Duan and Pettie [8, Section 1].

The main goal of the present work is to present a polylog-energy, polynomial-time distributed algorithm that computes a maximal matching in the network graph. The term maximal here indicates that the matching intersects every edge of the graph, and therefore cannot be augmented without first removing edges. It is well-known that the cardinality of any maximal matching is within a factor two of the largest (or maximum) matching.

▶ **Theorem 1.** *Let G be any graph on at most n vertices, of maximum degree at most Δ. Then Algorithm 1 always terminates in $O(\Delta \log n)$ timesteps, at which point each node knows its partner in a matching, M. Furthermore,*

$$\mathbb{P}\left(M \textit{ is a maximal matching and every node used energy } \leq 20C\log^2 n\right) \geq 1 - \frac{1}{n^2}.$$

Observe that the per-node energy use is polylog(n), which obviously cannot be improved by more than a polylog factor. Moreover, the time complexity bound, $O(\Delta \log n)$, is also nearly optimal, when one considers that G could contain a clique of size Δ, in which case, in order for all the nodes in that clique to get even one chance to send a message and have it received by the other nodes in the clique, there must be at least Δ timesteps, since our model does not allow a node the possibility to receive two or more messages in a single round. To put this another way, when Δ is small, a high degree of parallelism is possible, which our algorithm exploits; but, when Δ is large, there exist graphs for which this parallelism is impossible.

1.1 Application: Neighbor Assignment

One possible motivation for finding large matchings, apart from their intrinsic mathematical interest, comes from the desire to **back up data in case of node failures**. Suppose we had a perfect matching (that is, one whose edges contain every node) on the n nodes of our network. Then the matching could be viewed as pairing each node with a neighboring node that could serve as its backup device. This would ensure that each device has a load of one node to back up, and that each node is directly adjacent to its backup device.

Since perfect matchings are not always available, we consider a more general scheme, in which each node is assigned one of its neighbors to be its backup device, but we allow for loads greater than one. Such a function can be visualized as a directed graph, with a directed edge from each node to its backup device. In this case, each node has out-degree 1, and load equal to its in-degree. We would like to minimize the maximum load over all vertices.

In Section 6, we will show that, if one is willing to accept a maximum load that is $O(\log n)$ times the optimum, this problem can be simply reduced to the maximal matching problem. In light of our main result, this means that, if there exists a neighbor assignment with polylog(n) maximum load, then we can find one on a radio network, while using only polylog(n) energy.

1.2 Techniques

Our matching algorithm can be thought of as a distributed and low-energy version of the following greedy, centralized algorithm. Randomly shuffle the m edges. Then, processing the edges in order, accept each edge that is disjoint from all previous edges. Note that this always results in a maximal matching.

To make this into a distributed algorithm, we make each node, in parallel, try to establish contact with one of its unmatched neighbors to form an edge. Since a node can only receive a message successfully if exactly one of its neighbors is sending, we limit the probability for each node to participate in a given round, by setting a participation rate that is, with high probability, at most the inverse of the maximum degree of the residual graph induced by the unmatched nodes. It turns out that this can be accomplished using a set schedule, where the participation rate is a function of the amount of elapsed time.

The main technical obstacle in the analysis is proving that the maximum degree of the graph decreases according to schedule (or faster). This is achieved by noting that, if not, the first vertex to have its degree exceed the schedule would have to have been failed to be paired by our algorithm, despite going through a long sequence of consecutive rounds in which its chance to be paired was relatively high.

1.3 Related Work

Multi-hop radio network models have a long history, going back at least to work in the early 1990's by Bar-Yehuda, Goldreich, Itai [1, 2] among others. The particular model of energy-aware radio computation we are using was introduced by Chang et al. [4].

A recent result by Chatterjee, Gmyr, and Pandurangan [6] considered the closely related problem of Maximal Independent Set in another model, called the "Sleeping model." Although it has some interesting similarities to our work, there are several important differences. Firstly, we note that although matchings of G are nothing more than independent sets on the line graph of G, in distributed computing, we cannot just convert an algorithm designed to run on the line graph of G into an algorithm to run on G. Secondly, we note that the Sleeping model is based on the CONGEST model, and so, when a node is awake, it is allowed to send a different message to each of its neighbors at a unit cost. By contrast, in our model, one node can only send one message in a timestep, and it may collide with messages sent by other nodes.

Moscibroda and Wattenhofer [12] considered the problem of finding a Maximal Independent Set in a radio network. Their work also has some interesting similarities to ours, although they are assuming a unit-disk topology, and listening for messages is free in their model. On the other hand, their algorithm works even when the nodes wake up asynchronously at the start of the algorithm.

2 Preliminaries

2.1 Matchings

A *matching* is a subset of the edges of a graph G, such that no two of the edges share an endpoint. We say a matching is *maximum* if it has at least as many edges as any other matching for G. We say a matching is *maximal* if it is not contained in a larger matching for G. Equivalently, a matching is maximal if every edge of G shares at least one endpoint with an edge from the matching.

For $\alpha > 1$, we say a matching is α-approximately maximum if its cardinality is at least $1/\alpha$ times the cardinality of a maximum matching. It is an immediate consequence of the definitions that any maximal matching is 2-approximately maximum.

2.2 Low-Energy Radio Networks

We work in the Radio Network model, where we have a communication network on an underlying graph G. Each node in G is an identical processor equipped with a transmitter and receiver to communicate with other nodes. There is an edge between nodes u and v in the graph if u and v are within transmission range of each other. Time is divided into discrete timesteps. In each time step a processor can choose to do one of three actions: transmit, listen, or sleep. A message travels from a node u to a neighbor v of u at time t if

- u decides to transmit at time t,
- v decides to listen at time t and
- No other neighbor of v decides to transmit at time t.

Thus when a node u decides to send a message, that message is heard by **all** of its neighbors who happen to be listening, and for whom none of their other neighbors are sending.

So what happens if node v happens to listen and more than one of its neighbors is sending a message? There are several different models for this situation. In the most permissive of these, the LOCAL model, v receives all the messages sent by its neighbors. As already specified, we are not working in this model. A more restrictive model is the Collision Detection model (CD) where, when a listening node does not receive a message, it can can tell the difference between silence (no neighbors sending) and a collision (more than one neighbor sending). Our results hold in the even more restrictive “No Collision Detection” model, where a listening node receives a message when exactly one of its neighbors is sending, but it cannot distinguish between silence and colliding messages.

What about message sizes? The LOCAL model allows nodes to send messages of arbitrary size in a single timestep. In our work we assume that the messages sent in a single timestep are of size that is logarithmic in the size of the network. That is, when there are n nodes, the message sizes are $O(\log n)$.

We measure the cost of our algorithms in terms of their energy usage. We assume that a node incurs a cost of 1 energy unit each time that it decides to send or listen. When the node is sleeping there is no energy cost. We also assume that local computation is free. The goal of energy aware computation is to design algorithms where the nodes can schedule sleep and communication times so that the energy expenditure is small, ideally polylog(n), without compromising the time complexity too much, i.e., the running time remains poly(n).

3 Notation

3.1 The network

As mentioned earlier, $G = (V, E)$ is the graph defining our radio network. We denote $n = |V|$, and refer to the nodes as "processors." Although the processors are identical, and run identical code, we will assume each node has a unique ID that it knows and uses as its "name" in communication. We make the standard observation that, if each node were to generate an independently random string of $C \log n$ bits as its ID, the probability that all n nodes have distinct IDs is at least $1 - 1/n^{C-1}$, which can be made overwhelmingly likely.

When we present our pseudocode, it will be written from the perspective of a single processor. However, most of our analysis will be written from the "global" perspective of the entire graph.

3.2 Measuring time

To begin with, we define two units of time that will be used throughout the paper. The smaller unit of time is called a *timestep*, and refers to the basic time unit of our radio network model: in each timestep the nodes that choose to transmit are allowed to send a single message.

The larger unit of time, is called a *round*, and consists of three timesteps of the form $3t - 2, 3t - 1, 3t$, where $1 \le t \le t_{\max}$ is the round number. As shall be seen, rounds have the property that at the end of each round, the aggregate state of the network *encodes a matching*. More precisely, each node has a variable, `partner`, and at the end of each round, this variable is either the ID of one of its neighbor nodes, or has the value `null`; moreover, whenever, at the end of a round, $\texttt{partner}(v) = w \neq \texttt{null}$, we also have $\texttt{partner}(w) = v$.

3.3 The Evolving Matching

For $t \ge 0$, we denote by $M(t)$ the matching encoded by the network at the end of round t; this is a random variable whose value is always a pairwise disjoint set of edges of the graph. As discussed earlier, $M(t)$ is well defined because, at the end of every round, all vertices have a mutually consistent view of whom they are paired to.

We mention that our matching will be non-decreasing over time, that is, for all $t < t'$, $M(t) \subseteq M(t')$ with probability one. Next we define a few useful notations, all defined in terms of G and $M(t)$.

By $G(t)$, we denote the *residual graph at the end of round* t; that is, the graph induced by all unmatched vertices of G. Thus $G(t) = (V(t), E(t))$, where $V(t) = V \setminus \bigcup M(t)$ and $E(t) = E \cap \binom{V(t)}{2}$. Similarly, for a vertex v, we will refer to its residual neighbor set, $N(v, t) = N(v) \cap V(t)$, and its residual degree, $d(v, t) = |N(v, t)|$. We will denote the maximum degree in the residual graph by $\Delta(t) = \max\limits_{v \in V(t)} d(v, t)$, taking this value to be zero if $V(t)$ is empty.

4 Maximal Matching Algorithm

The basic idea of our algorithm is, starting with the empty matching, to greedily add disjoint edges until a maximal matching is achieved. The challenge is to keep each node's energy cost low. We achieve this by having nodes wake up at random times, and try to recruit one of their neighbors to pair with them. If this succeeds, and is not hampered by additional, redundant, neighbors that also happen to wake up, then an edge is added to the matching.

To ensure that both endpoints of the edge agree about who they are paired with, the nodes execute a three-step "handshake" protocol, with the property that, if it succeeds, both nodes know that the other node has only been in communication with them, and was not, for instance, trying to form an edge with another, different, endpoint.

To keep the energy costs low, it is essential that nodes wake up with approximately the correct frequency. If the rate is too high (significantly greater than the inverse of the degree), too many nodes will wake up at once, causing collisions. Even if we get around these collisions by some device, having too many nodes wake up at once seems likely to lead to excessive energy consumption, since at most one neighbor of a node can get a message through in a single round.

If the rate is too low (significantly less than the inverse of the degree), then too few nodes will wake up at once, again leading to an excessive waste of energy, since a node whose neighbors are all asleep cannot get anywhere.

We define the "participation rate," $r(t)$ by the function

$$r(t) = \frac{3C\log(n)}{4C\log(n) + t_{\max} - t}$$

where $t_{\max} = C\Delta\log(n)$. This function gives a schedule for gradually raising the participation rate from $\Theta(1/\Delta)$ up to $\Theta(1)$. The constant C will be specified in the proof of Theorem 1.

Algorithm 1 Main Algorithm: A Low-Energy Distributed implementation of Greedy Maximal Matching in a Radio Network.

```
t ← 1
partner ← null
while partner = null and $t \le t_{\max}$ do
    Sample $x$ uniformly from $[0, 1]$.
    if $x \le r(t)/2$ then
        Do RECRUIT_PROTOCOL this round.
    else if $r(t)/2 < x \le r(t)$ then
        Do ACCEPT_PROTOCOL this round.
    else
        Sleep this round.
    end if
    $t \leftarrow t + 1$
end while
Sleep for the remaining $t_{\max} - t$ rounds.
```

Algorithm 2 RECRUIT_PROTOCOL: Try to form an edge as initial sender.

```
At timestep 1, Send $my\_ID$                          ▷ "My name is $my\_ID$ and I am available"
At timestep 2, Listen
if message received then
    Interpret the message as an ordered pair of integers $(x, y)$
    if $x = my\_ID$ then                                        ▷ Match found
        partner ← $y$
        At timestep 3, send $(x, y)$                           ▷ "$x$ and $y$ are paired"
    end if
else
    Sleep for timestep 3.
end if
```

Algorithm 3 ACCEPT_PROTOCOL: Try to form an edge as initial listener.

```
At timestep 1, Listen
if message received then
    Interpret the message as an integer $x$
    At timestep 2, Send $(x, my\_ID)$              ▷ "Hello, lets match up, $x$ and $my\_ID$"
    At timestep 3, Listen
    if message received then
        Interpret the message as an ordered pair of integers $(x, y)$
        if $(y == my\_ID)$ then                            ▷ $x$ and $y$ are matched
            partner ← $x$
        end if
    end if
end if
Sleep for any timesteps remaining in the round.
```

5 Maximal Matching Analysis

▶ **Lemma 2.** *With probability one, at the end of every round $t \geq 0$, the* `partner` *variables of the n nodes encode a well-defined matching $M(t)$.*

Proof. Initially, all the vertices are unmatched, with `null` partners, so $M(0) = \emptyset$. Later, we observe that the only circumstances under which the `partner` variables have their values reassigned is when a vertex v has chosen to participate in that round as recruiter, a neighboring vertex w has chosen to participate in that round as accepter, and furthermore, both v and w receive a message each time they Listen during their respective protocols. Since a message is received if and only if exactly one neighbor Sends in that timestep, the messages v receives must come from w, and vice-versa. Therefore v stores the id of w in its partner variable, and vice-versa.

Furthermore, since v and w would not have participated in round t unless their `partner` variables were both `null` beforehand, we know by induction that no other vertices have v or w as their partners. Since this applies for all vertices and all rounds, the pairing is one-to-one, as desired. ◀

▶ **Lemma 3.** *Let $t \geq 1$, let $\{v, w\} \in E$, and let $X_{v,w,t}$ be the indicator random variable for the event that v and w get matched to each other in round t. Then*

$$\mathbb{E}\left(X_{v,w,t} \mid M(t-1) \right) \geq \frac{r(t)^2}{2} \left(1 - r(t)\right)^{\Delta(t-1)-1} \mathbf{1}(v, w \in V(t-1)) \, .$$

Proof. In order for an edge to form between v and w in round t, it is necessary and sufficient for the following four events all to occur:

$E_0 = \{v, w \in V(t-1)\}$,

$E_1 = \{v, w$ both participate in round t $\}$,

$E_2 = \{$exactly one of v, w participates as recruiter, the other as accepter, in round $t\}$,

$E_3 = \{E_2$, and v and w receive each others messages without any collision$\}$.

Note that $E_3 \subset E_2 \subset E_1 \subset E_0$. For $0 \leq i \leq 3$, let $X_i = \mathbf{1}(E_i)$. We now compute expectations, conditioned on the matching at the end of the previous round. We will prove, below, that

$$\mathbb{E}\left(X_1 \mid M(t-1) \right) = r(t)^2 X_0, \tag{1}$$

$$\mathbb{E}\left(X_2 \mid X_1, M(t-1) \right) = \frac{1}{2} X_1, \tag{2}$$

$$\mathbb{E}\left(X_3 \mid X_2, X_1, M(t-1) \right) \geq \left(1 - r(t)\right)^{\Delta(t-1)-1} X_2. \tag{3}$$

It follows by the Law of Total Probability that

$$\mathbb{E}\left(X_3 \mid M(t-1) \right) \geq \frac{r(t)^2}{2} \left(1 - r(t)\right)^{\Delta(t-1)-1} X_0,$$

which is equivalent to the statement of the lemma, noting that $X_3 = X_{v,w,t}$ and $X_0 = \mathbf{1}(v, w \in V(t-1))$.

Now, (1) and (2) follow immediately from the definitions of E_1, E_2, and the fact that every vertex in $V(t-1)$ has probability $r(t)/2$ to participate as recruiter in round t, and the same probability to participate as accepter. To establish (3), we first note that, assuming E_2 occurs, for E_3 to occur it is sufficient that no other neighbor of w decides to participate in round t in the same role as v, and no other neighbor of v decides to participate in the same role as w. Since each vertex makes its participation decision independently, this probability equals

$$\left(1 - r(t)\right)^A \left(1 - \frac{r(t)}{2}\right)^B \geq \left(1 - r(t)\right)^{A + B/2},$$

where $A = |N(v, t-1) \cap N(w, t-1)|$ and $B = |N(v, t-1) \oplus N(w, t-1) \setminus \{v, w\}|$. But since $2A + B = |N(v, t-1) \setminus \{w\}| + |N(w, t-1) \setminus \{v\}| \leq 2(\Delta(t-1) - 1)$, this establishes (3), completing the proof. ◀

We now introduce a random variable that will be used crucially in the remainder of our analysis.

▶ **Definition 4.** *Let v be a vertex, and $1 \leq t \leq t_{\max}$. $Z(v, t)$ denotes the indicator random variable for the event $\{d(v, t) \geq 1/(2r(t))$ and $\Delta(t-1) \leq 1/r(t)\}$.*

This definition is motivated by the observation that, if t were minimal such that there exists a v such that $d(v, t) \geq 1/(2r(t))$, then $Z(v, t)$ would have to have value 1 for many immediately preceding rounds, which we can prove is highly unlikely.

▶ **Lemma 5.**

$$\mathbb{E}\left(Z(v,t) \mid M(t-1) \right) \le 1 - \frac{r(t)}{4e^2}.$$

Proof. Recall that, by definition, $Z(v,t) = 1$ if and only if: $v \in V(t)$ and $d(v,t) \geq 1/2r(t)$ and $\Delta(t-1) \leq 1/r(t)$. Now, since the degrees at time $t-1$ are determined by $M(t-1)$, and there is nothing to prove when the conditional information implies $Z(v,t)$ is identically zero, we may assume $d(v,t-1) \geq 1/2r(t)$ and $\max_w d(w,t-1) \leq 1/r(t)$.

Also, note that $v \notin V(t)$ will occur if and only if $X_{v,w,t} = 1$ for some $w \in N(v,t-1)$. Since these events are disjoint, we may sum their probabilities, obtaining

$$\begin{aligned}\mathbb{E}\left(Z(v,t) \mid M(t-1) \right) &\le 1 - \sum_w \mathbb{E}\left(X_{v,w,t} \mid M(t-1) \right) \\ &\le 1 - d(v,t-1)\frac{r(t)^2}{2}\left(1-r(t)\right)^{\Delta(t-1)-1} && \text{by Lemma 3.} \\ &\le 1 - d(v,t-1)\frac{r(t)^2}{2}\left(1-r(t)\right)^{1/r(t)}\end{aligned}$$

Additionally, since $Z(v,t) = 0$ unless $d(v,t) \geq 1/2r(t)$, we have

$$\begin{aligned}\mathbb{E}\left(Z(v,t) \mid M(t-1) \right) &\le 1 - \frac{1}{2r(t)}\frac{r(t)^2}{2}\left(1-r(t)\right)^{1/r(t)} \\ &\le 1 - \frac{r(t)}{4}\left(1-r(t)\right)^{1/r(t)} \\ &\le 1 - \frac{r(t)}{4e^2}.\end{aligned}$$

Here the last inequality follows because on the interval $[0, 3/4]$ we have $1 - x > e^{-2x}$, and $r(t) \leq 3/4$ for all t. ◀

We are now ready to present the proof of Theorem 1.

Proof of Theorem 1. We focus on the probability that M is maximal. If not, then there exists a vertex whose degree is positive after round $t_{\max}$. Since $r(t_{\max}) = 3/4$, it follows that the set

$$S = \left\{ (v,t) \colon v \in V, 0 < t \leq t_{\max}, d(v,t) \geq \frac{3}{4r(t)} \right\}$$

is non-empty. Let t be the smallest time such that there exists a vertex v with $(v,t) \in S$.

Let I be the set of all times $t' \leq t$ such that $2r(t)/3 < r(t') \leq r(t)$. By our assumption that t is minimal, we have

$$d(w, t'-1) < \frac{3}{4r(t'-1)} < \frac{1}{r(t')}$$

for all vertices $w \in V$ and times $t' \in I$. Hence $\Delta(t'-1) < 1/r(t')$ for all $t' \in I$. On the other hand, since $d(v,t')$ is a non-increasing function of t', we also have

$$d(v,t') \geq d(v,t) \geq \frac{3}{4r(t)} > \frac{1}{2r(t')},$$

for all $t' \in I$, where the last inequality is by the definition of I. It follows that

for all $t' \in I$, $Z(v,t') = 1$.

Let $\mathcal{E}_{v,t}$ be the above event, *i.e.,* that $\prod_{t' \in I} Z(v,t') = 1$. We want to compute the probability of $\mathcal{E}_{v,t}$. Let t_0 be the smallest time in I. Then

$$\begin{aligned}
\mathbb{P}\left(\mathcal{E}_{v,t}\right) &= \mathbb{E}\left(\prod_{t'=t_0}^{t} Z(v,t')\right) \\
&= \mathbb{E}\left(\mathbb{E}\left(\prod_{t'=t_0}^{t} Z(v,t') \,\middle|\, M(t-1)\right)\right) && \text{by the Law of Total Expectation} \\
&= \mathbb{E}\left(\left(\prod_{t'=t_0}^{t-1} Z(v,t')\right) \mathbb{E}\left(Z(v,t) \mid M(t-1)\right)\right) && (*) \\
&\le \mathbb{E}\left(\prod_{t'=t_0}^{t-1} Z(v,t')\right)\left(1 - \frac{r(t)}{4e^2}\right) && \text{by Lemma 5}
\end{aligned}$$

Line (*) follows since $M(t-1)$ determines $Z(v,t_0), \ldots, Z(v,t-1)$. By repeating the steps from using the Law of Total Expectation to applying Lemma 5 on the remaining expectation of a product, we see by induction that

$$\begin{aligned}
\mathbb{P}\left(\mathcal{E}_{v,t}\right) &\le \prod_{t' \in I}\left(1 - \frac{r(t')}{4e^2}\right) \\
&\le \exp\left(-\frac{1}{4e^2}\sum_{t' \in I} r(t')\right) && \text{since for all } x,\ 1 - x \le e^{-x}
\end{aligned}$$

To get a handle on the expression on the right hand side, we need a lower bound on the sum of the participation rates. Let $t^* \in \mathbb{R}$ be such that $r(t^*) = \frac{2}{3}r(t)$. Then $t_0 - 1 < t^* \le t_0$, and we have

$$\begin{aligned}
\sum_{t'=t_0}^{t} r(t') &\ge \int_{t_0-1}^{t} r(t')\mathrm{d}t' && \text{upper Riemann sum} \\
&\ge \int_{t^*}^{t} r(t')\mathrm{d}t' \\
&\ge \int_{t^*}^{t} \frac{3C\log n}{4C\log n + t_{\max} - t'}\mathrm{d}t' \\
&= 3C\log n[\log(4C\log n + t_{\max} - t^*) - \log(4C\log n + t_{\max} - t)] \\
&= 3C\log n \log\left(\frac{r(t)}{r(t^*)}\right) \\
&= 3C\log n\log(3/2)
\end{aligned}$$

Plugging this back into the probability calculation,

$$\begin{aligned}
\mathbb{P}\left(\mathcal{E}_{v,t}\right) &\le \exp\left(-\frac{1}{4e^2}\sum_{t' \in I} r(t')\right) \\
&\le \exp\left(-\frac{3C\log n\log(3/2)}{4e^2}\right) \\
&\le \frac{1}{n^{4+\epsilon}}.
\end{aligned}$$

where the last inequality holds for suitably large values of C, *e.g.,* when $C = 100$.

Taking a union bound over the $nt_{\max} = O(n^2 \log n)$ events $\mathcal{E}_{v,t}$ completes the proof of correctness.

The upper bound on energy use comes from a simple analysis of the number of rounds each vertex participates in. Clearly, the energy use is at most 3 times the number of rounds the vertex participates in, which is at most the number of heads that would be flipped in $t_{\max}$ independent coin flips, with probabilities of heads $r(1), r(2), \ldots, r(t_{\max})$. Hence the expected energy use is at most the sum, which we calculated above as approximately $3 \cdot 3C \log n \log \left(\frac{r(t_{\max})}{r(0)}\right)$, which is at most $9C(\log n)^2 = O(\log^2 n)$. Chernoff's bound, together with with a union bound over the n vertices, implies the high-probability upper bound on expected energy cost. ◀

6 Neighbor Assignment Functions

Motivated by the problem of assigning nodes to backup data from their neighbors in a sensor network, we introduce the following definition. As we shall see later, it is extremely closely connected to the established concept of matching covering number.

▶ **Definition 6.** *Given graph $G = (V, E)$, a* neighbor assignment function *(NAF) is a function $f : V \to V$ such that for all $v \in V$, $\{v, f(v)\} \in E$. Equivalently, we may think of this as an oriented subgraph of G, in which each vertex has out-degree 1. The* load *of the assignment is the maximum in-degree of this digraph. Equivalently, load is $\max_{v \in V} |f^{-1}(v)|$. The* minimum NAF load *of G is the minimum load among all NAFs for G.*

Note. In the case when G is bipartite, NAFs are also known as "semi-matchings." (See, for example, [9, 7].) However, since we are particularly concerned with the non-bipartite case, we preferred to introduce a different term.

In the context of backing up data, we think of the assigned node $f(v)$ as the node who will store a backup copy of v's data. Our goal for this section is to find a NAF whose load is small. In the energy-aware radio network setting, we also want to ensure that the per-node energy use is small.

Our next result establishes a close connection between the load of the best NAF for a graph and the minimum number of matchings needed to cover all of its vertices.

▶ **Definition 7.** *The* matching cover number *of a graph G, denoted $\mathrm{mc}(G)$, is the minimum integer k such that there exists a set of k matchings of G, whose union contains every vertex of G.*

▶ **Theorem 8.** *For every graph G, the minimum NAF-load of G equals the matching cover number of G, unless the NAF-load of G equals 1. If the NAF-load of G equals 1, the matching cover number of G can be 1 or 2.*

Proof. Suppose $V = V(G)$ is covered by the union of matchings $M_1, \ldots, M_L$. Then assigning each vertex v to its partner in the first matching that contains v is an NAF with maximum load at most L. This establishes that the NAF-load is always at most the matching covering number.

Before we begin the proof for the reverse implication, we make the following general observation about digraphs with out-degree 1. By considering the unique walk obtained by starting at any vertex v, and repeatedly following the edge $\{v, f(v)\}$, we can see that each weakly connected component consists of one oriented cycle (of length ≥ 2), together with one or more "tributary" trees, each rooted at a node of this cycle, and oriented towards that root. See Figure 1.

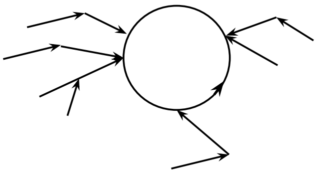


Figure 1 Sketch of the digraph of a NAF with only one component. Each component can be seen as a directed cycle, along with zero or more "tributary trees."

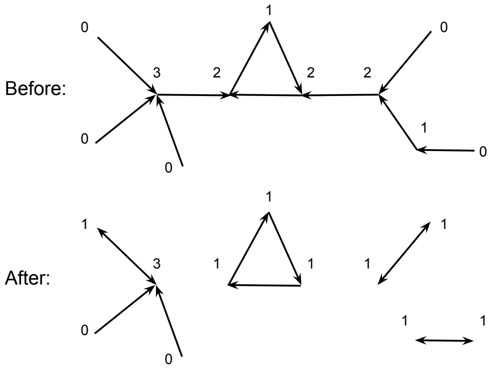


Figure 2 Example of the conversion of a given NAF by decreasing the number of load-zero nodes. The number by each node indicates its load. Note that, after the conversion, the maximum load did not increase, and the connected components of the NAF are now all directed cycles and/or stars with one bi-directed edge. (The components of size 2 are both.)

Now, if f has any leaf, that is, a node, v, whose load is zero, we can obtain a new NAF by reassigning $f(v)$ to point back to v. This increases the load at v to 1, decreases the load by 1 at $f(f(v))$, and does not change any other vertex loads. Repeated application of this rule to all leaves in turn, eventually leads to a NAF whose components are all either (a) directed cycles, which do not have any leaves, or (b) stars with one bi-directed edge. See Figure 2. In case (b), the component consists of one node, r, of in-degree $i \leq L$, i nodes, $x_1, \ldots, x_i$, each with an edge directed to r, and one edge from r to x_1.

It is easy to see that, for a directed cycle, whose edges are $e_1, \ldots, e_\ell$, a single matching consisting of the even edges, $e_2, e_4, \ldots,$, will cover all the vertices if ℓ is even, and all but one vertex if ℓ is odd. Therefore, one matching covers the component if ℓ is even, and two if ℓ is odd.

For the star with bi-directed edge, the maximum load equals the degree, d, of the center vertex. And a matching cover consists of the d single edges that make up the star.

In this way, we can build up our matching cover component by component, noting that if every component has a matching cover of size at most k, then so does the entire graph. Since the only case when our matching cover was smaller than the maximum load for the component was when $L = 1$, the proof is complete. ◀

Wang, Song, and Yuan [14] have given an $O(n^3)$-time centralized algorithm for finding the minimum number of matchings needed to cover a graph. In light of Theorem 8, their result implies an $O(n^3)$ algorithm for finding the minimum-load NAF for any graph.

In the distributed and low-energy setting, it is unlikely that we can achieve such an ambitious goal. For instance, a node cannot determine its exact degree without receiving that many messages successfully, which may require linear energy. Instead, we aim for the less ambitious goal of finding a NAF whose maximum load is well within our energy budget. Our next result shows that this is possible, assuming one exists.

Algorithm 4 Low-Energy Distributed algorithm to compute a NAF in a Radio Network.

```
Run our Maximal Matching algorithm on $G$.
For each edge $\{u, v\}$ in the matching, mark $u, v$ as assigned, and assign them to each other.
for $i \leftarrow 1$ to $k$ do
    Run the maximal matching algorithm on $G$, modified so that only unassigned nodes are allowed to recruit, and only assigned nodes are allowed to accept.
    For each edge $\{u, v\}$ in the matching, mark $u, v$ as assigned, and (re-)assign them to each other.
end for
```

▶ **Theorem 9.** *Suppose G has a NAF with maximum load L. Then Algorithm 4, run with parameter $k = O(L \log n)$, will, with high probability, output a NAF with maximum load $O(L \log n)$. Its per-vertex energy usage will be $O(L \operatorname{polylog} n)$.*

Proof. Since G has a NAF with maximum load L, by Theorem 8, it has a vertex cover by L matchings. This implies that the maximum matching covers at least N/L vertices. Hence every maximal matching covers at least $N/2L$ vertices. So the first call to the maximal matching algorithm will assign neighbors to at least $N/2L$ vertices.

In subsequent rounds, the modification to the maximal matching algorithm has the effect of making it run on the bipartite graph where the bipartition is into the assigned and unassigned vertices. Now, consider the vertex cover by the L matchings in G. By pigeonhole principle, at least one of these matchings, M, must cover at least a $1/L$ fraction of the unassigned vertices. Since the first matching was maximal, no edges in G have both endpoints unassigned; therefore, M is a matching within the bipartite graph being fed into our maximal matchings algorithm. Therefore, the maximal matching that is found must cover at least a $1/2L$ fraction of the unassigned vertices. It follows that after k iterations, at most

$$\left(1 - \frac{1}{2L}\right)^k n \leq e^{-k/2L} n$$

nodes will remain unassigned.

Setting $k = 2L \log(n)$ gives a good chance that no nodes remain unassigned, and so the algorithm produces a NAF. Clearly, since the digraph for the NAF is contained within the union of k matchings, this is an upper bound on the maximum load. Since each run of the maximal matching algorithm succeeds with probability $1 - O(1/n^2)$, a union bound establishes the high-probability bound.

More generally, if a partial NAF exists, that covers a $(1 - \varepsilon)$ fraction of the nodes, with maximum load L, and we run Algorithm 4 with $k = 2L \log(1/\delta)$, it should, with high probability, produce a partial NAF that covers at least $(1 - \varepsilon)(1 - \delta)n$ nodes, and has maximum load k ◀

References

1 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. Efficient emulation of single-hop radio network with collision detection on multi-hop radio network with no collision detection. *Distributed Computing*, 5(2):67–71, 1991.

2 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.

3 Matthew Barnes, Chris Conway, James Mathews, and DK Arvind. Ens: An energy harvesting wireless sensor network platform. In *2010 Fifth International Conference on Systems and Networks Communications*, pages 83–87. IEEE, 2010.

4 Yi-Jun Chang, Varsha Dani, Thomas P Hayes, Qizheng He, Wenzheng Li, and Seth Pettie. The energy complexity of broadcast. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 95–104, 2018.

5 Yi-Jun Chang, Varsha Dani, Thomas P Hayes, and Seth Pettie. The energy complexity of BFS in radio networks. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 273–282, 2020.

6 Soumyottam Chatterjee, Robert Gmyr, and Gopal Pandurangan. Sleeping is efficient: Mis in $o(1)$-rounds node-averaged awake complexity. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 99–108, 2020.

7 Andrzej Czygrinow, Michał Hanćkowiak, Edyta Szymańska, and Wojciech Wawrzyniak. On the distributed complexity of the semi-matching problem. *Journal of Computer and System Sciences*, 82(8):1251–1267, 2016.

8 Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1–23, 2014.

9 Nicholas JA Harvey, Richard E Ladner, László Lovász, and Tami Tamir. Semi-matchings for bipartite graphs and load balancing. *Journal of Algorithms*, 59(1):53–78, 2006.

10 Wendi R. Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, pages 10–pp. IEEE, 2000.

11 Dénes König. Über graphen und ihre anwendung auf determinantentheorie und mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916.

12 Thomas Moscibroda and Roger Wattenhofer. Maximal independent sets in radio networks. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 148–157, 2005.

13 Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, pages 364–369. IEEE, 2005.

14 Xiumei Wang, Xiaoxin Song, and Jinjiang Yuan. On matching cover of graphs. *Mathematical Programming*, 147(1):499–518, 2014.

The Canonical Amoebot Model: Algorithms and Concurrency Control

Joshua J. Daymude
Biodesign Center for Biocomputing, Security and Society,
Arizona State University, Tempe, AZ, USA

Andréa W. Richa
School of Computing and Augmented Intelligence, Arizona State University, Tempe, AZ, USA

Christian Scheideler
Department of Computer Science, Universität Paderborn, Germany

Abstract

The *amoebot model* abstracts active programmable matter as a collection of simple computational elements called *amoebots* that interact locally to collectively achieve tasks of coordination and movement. Since its introduction (SPAA 2014), a growing body of literature has adapted its assumptions for a variety of problems; however, without a standardized hierarchy of assumptions, precise systematic comparison of results under the amoebot model is difficult. We propose the *canonical amoebot model*, an updated formalization that distinguishes between core model features and families of assumption variants. A key improvement addressed by the canonical amoebot model is *concurrency*. Much of the existing literature implicitly assumes amoebot actions are isolated and reliable, reducing analysis to the sequential setting where at most one amoebot is active at a time. However, real programmable matter systems are concurrent. The canonical amoebot model formalizes all amoebot communication as message passing, leveraging adversarial activation models of concurrent executions. Under this granular treatment of time, we take two complementary approaches to *concurrent algorithm design*. Using *hexagon formation* as a case study, we first establish a set of *sufficient conditions* for algorithm correctness under any concurrent execution, embedding concurrency control directly in algorithm design. We then present a *concurrency control framework* that uses locks to convert amoebot algorithms that terminate in the sequential setting and satisfy certain conventions into algorithms that exhibit equivalent behavior in the concurrent setting. Together, the canonical amoebot model and these complementary approaches to concurrent algorithm design open new directions for distributed computing research on programmable matter.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Concurrency; Theory of computation → Self-organization

Keywords and phrases Programmable matter, self-organization, distributed algorithms, concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.20

Related Version *Full Version*: `https://arxiv.org/abs/2105.02420` [17]

Funding *Joshua J. Daymude*: NSF (CCF-1733680), U.S. ARO (MURI W911NF-19-1-0233), and the ASU Biodesign Institute.
Andréa W. Richa: NSF (CCF-1733680) and U.S. ARO (MURI W911NF-19-1-0233).
Christian Scheideler: DFG Project SCHE 1592/6-1.

Acknowledgements We thank Nicola Santoro and Paola Flocchini for their constructive feedback and Kristian Hinnenthal for his contributions to a preliminary version of this work.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 20; pp. 20:1–20:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The vision of *programmable matter* is to realize a material that can dynamically alter its physical properties in a programmable fashion, controlled either by user input or its own autonomous sensing of its environment [38]. Towards a formal characterization of the minimum capabilities required by individual modules of programmable matter to achieve a given system behavior, many abstract models have been proposed over the last several decades [3, 8, 9, 10, 29, 35, 36, 37, 39]. We focus on the *amoebot model* [16, 19] which is motivated by micro- and nano-scale robotic systems with strictly limited computational and locomotive capabilities [32, 33, 34, 40, 41]. The amoebot model abstracts active programmable matter as a collection of simple computational elements called *amoebots* that utilize local interactions to collectively achieve tasks involving coordination, movement, and reconfiguration. Since its introduction in 2014, the amoebot model has been used to study both fundamental problems – such as leader election [5, 15, 23, 25, 28, 31] and shape formation [7, 20, 21, 24, 25] – as well as more complex behaviors including object coating [13, 22], convex hull formation [14], bridging [2], spatial sorting [6], and fault tolerance [18, 27].

With this growing body of amoebot model literature, it is evident that the model has evolved – and, to some extent, fractured – during its lifetime as assumptions were updated to support individual results, capture more realistic settings, or better align with other models of programmable matter. This makes it difficult to conduct any systematic comparison between results under the amoebot model (see, e.g., the overlapping but distinct features used for comparison of leader election algorithms in [5] and [28]), let alone between amoebot model results and those of related models (e.g., those from the established *autonomous mobile robots* literature [29]). To address the ways in which the amoebot model has outgrown its original rigid formulation, we propose the *canonical amoebot model* that includes a standardized, formal hierarchy of assumptions for its features to better facilitate comparison of its results. Moreover, such standardization will more gracefully support future model generalizations by distinguishing between core features and assumption variants.

A key area of improvement addressed by the canonical amoebot model is *concurrency*. The original model treats concurrency at a high level, implicitly assuming an isolation property that prohibits concurrent amoebot actions from interfering with each other. Furthermore, amoebots are usually assumed to be *reliable*; i.e., they cannot crash or exhibit Byzantine behavior. Under these simplifying assumptions, most existing algorithms are analyzed for correctness and runtime as if they are executed *sequentially*, with at most one amoebot acting at a time. Notable exceptions include the recent work of Di Luna et al. [24, 25, 27] that adopt ideas from the "look-compute-move" paradigm used in autonomous mobile robots to bring the amoebot model closer to a realistic, concurrent setting. Our canonical amoebot model furthers these efforts by formalizing all communication and cooperation between amoebots as message passing while also addressing the complexity of potential conflicts caused by amoebot movements. This careful formalization allows us to use standard adversarial activation models from the distributed computing literature to describe concurrency [1].

This fine-grained treatment of concurrency in the canonical amoebot model lays the foundation for the design and analysis of *concurrent amoebot algorithms*. Concurrency adds significant design complexity, allowing concurrent amoebot actions to mutually interfere, conflict, affect outcomes, or fail in ways far beyond what is possible in the sequential setting. As a tool for controlling concurrency, we introduce a Lock operation in the canonical amoebot model enabling amoebots to attempt to gain exclusive access to their neighborhood.

We then take two complementary approaches to concurrent algorithm design in the canonical amoebot model: a direct approach that embeds concurrency control directly into the algorithm's design without requiring locks, and an indirect approach that relies on the LOCK operation to mitigate issues of concurrency. In the first approach, we establish a set of *general sufficient conditions* for amoebot algorithm correctness under any adversary – sequential or asynchronous, fair or unfair – using the *hexagon formation problem* (see, e.g., [16, 20]) as a case study. Our Hexagon-Formation algorithm demonstrates that locks are not necessary for correctness even under an unfair, asynchronous adversary. However, this algorithm's asynchronous correctness relies critically on its actions succeeding despite any concurrent action executions, which may be a difficult property to obtain in general.

For our second approach, we present a *concurrency control framework* using the LOCK operation that, given an amoebot algorithm that terminates under any sequential execution and satisfies some basic conventions, produces an algorithm that exhibits equivalent behavior under any asynchronous execution. This framework establishes a general design paradigm for concurrent amoebot algorithms: one can first design an algorithm with correct behavior in the simpler sequential setting and then, by ensuring it satisfies our framework's conventions, automatically obtain a correct algorithm for the asynchronous setting. The convenience of this approach comes at the cost of the full generality of the canonical amoebot model which is limited by the framework's conventions.

Our Contributions. We summarize our contributions as follows.

- The *canonical amoebot model*, an updated formalization that treats amoebot actions at the fine-grained level of message passing and distinguishes between core model features and hierarchies of assumption variants (Section 2).
- General *sufficient conditions* for amoebot algorithm correctness under any adversary and an algorithm for *hexagon formation* that satisfies these conditions (Section 3).
- A *concurrency control framework* that converts amoebot algorithms that terminate under any sequential execution and satisfy certain conventions into algorithms that exhibit equivalent behavior under any asynchronous execution (Section 4).

1.1 Related Work

There are many theoretical models of programmable matter, ranging from the non-spatial *population protocols* [3] and *network constructors* [35] to the tile-based models of *DNA computing* and *molecular self-assembly* [8, 36, 39]. Most closely related to the amoebot model studied in this work is the well-established literature on *autonomous mobile robots*, and in particular those using discrete, graph-based models of space (see Chapter 1 of [29] for a recent overview). Both models assume anonymous individuals that can actively move, lacking a global coordinate system or common orientation, and having strictly limited computational and sensing capabilities. In addition, stronger capabilities assumed by the amoebot model also appear in more recent variants of mobile robots, such as persistent memory in the $\mathcal{F}$*-state* model [4, 30] and limited communication capabilities in *luminous robots* [11, 12, 26].

There are also key differences between the amoebot model and the standard assumptions for mobile robots, particularly around their treatment of physical space, the structure of individuals' actions, and concurrency. First, while the discrete-space mobile robots literature abstractly envisions robots as agents occupying nodes of a graph – allowing multiple robots to occupy the same node – the amoebot model assumes *physical exclusion* that ensures each node is occupied by at most one amoebot at a time, inspired by the real constraints of self-organizing micro-robots and colloidal state machines [32, 33, 34, 40, 41]. Physical exclusion introduces conflicts of movement (e.g., two amoebots concurrently moving into the same space) that must be handled carefully in algorithm design.

Second, mobile robots are assumed to operate in *look-compute-move* cycles, where they take an instantaneous snapshot of their surroundings (look), perform internal computation based on the snapshot (compute), and finally move to a neighboring node determined in the compute stage (move). While it is reasonable to assume robots may instantaneously snapshot their surroundings due to all information being visible, the amoebot model – and especially the canonical version presented in this work – treats all inter-amoebot communication as asynchronous message passing, making snapshots nontrivial. Moreover, amoebots have *read and write* operations allowing them to access or update variables stored in the persistent memories of their neighbors that do not fit cleanly within the look-compute-move paradigm.

Finally, the mobile robots literature has a well-established and carefully studied hierarchy of *adversarial schedulers* capturing assumptions on concurrency that the amoebot model has historically lacked. In fact, other than notable recent works that adapt look-compute-move cycles and a semi-synchronous scheduler from mobile robots to the amoebot model [24, 25, 27], most amoebot literature assumes only sequential activations. A key contribution of our canonical amoebot model presented in this work is a hierarchy of concurrency and fairness assumptions similar in spirit to that of mobile robots, though our underlying message passing design and lack of explicit action structure require different formalizations.

2 The Canonical Amoebot Model

We introduce the *canonical amoebot model* as an update to the model's original formulation [16, 19]. This update has two main goals. First, we model all amoebot actions and operations using message passing, leveraging this finer level of granularity for a formal treatment of concurrency. Second, we clearly delineate which assumptions are fixed features of the model and which have stronger and weaker variants (Table 1), providing unifying terminology for future amoebot model research. Unless variants are explicitly listed, the following description of the canonical amoebot model details its core, fixed assumptions.

In the canonical amoebot model, programmable matter consists of individual, homogeneous computational elements called *amoebots*. Any structure that an amoebot system can form is represented as a subgraph of an infinite, undirected graph $G = (V, E)$ where V represents all relative positions an amoebot can occupy and E represents all atomic movements an amoebot can make. Each node in V can be occupied by at most one amoebot at a time. There are many potential variants with respect to space; the most common is the *geometric* variant that assumes $G = G_\Delta$, the triangular lattice (Figure 1a).

An amoebot has two *shapes*: CONTRACTED, meaning it occupies a single node in V, or EXPANDED, meaning it occupies a pair of adjacent nodes in V (Figure 1b). For a contracted amoebot, the unique node it occupies is considered to be its *head*; for an expanded amoebot, the node it has most recently come to occupy (due to movements) is considered its head and the other is its *tail*. Each amoebot keeps a collection of ports – one for each edge incident to the node(s) it occupies – that are labeled consecutively according to its own local *orientation*. An amoebot's orientation is persistent and depends on its *direction* – i.e., what it perceives as "north" – and its *chirality*, or sense of clockwise and counter-clockwise rotation. Different variants may assume that amoebots share one, both, or neither of their directions and chiralities (see Table 1); Figure 1c gives an example of the *common chirality* variant where amoebots share a sense of clockwise rotation but have different directions.

Table 1 Summary of assumption variants in the canonical amoebot model, each organized from most to least general. Variants marked with $*$ have been considered in existing literature, and variants marked with † are the focus of the algorithmic results in this work.

	Variant	Description
Space	General*	G is any infinite, undirected graph.
	Geometric*,†	$G = G_\Delta$, the triangular lattice.
Orientation	Assorted*,†	Assorted direction and chirality.
	Common Chirality*	Assorted direction but common chirality.
	Common Direction	Common direction but assorted chirality.
	Common	Common direction and chirality.
Memory	Oblivious	No persistent memory.
	Constant-Size*,†	Memory size is $\mathcal{O}(1)$.
	Finite	Memory size is $\mathcal{O}(f(n))$, some function of the system size.
	Unbounded	Memory size is unbounded.
Concurrency	Asynchronous†	Any amoebots can be simultaneously active.
	Synchronous*	Any amoebots can simultaneously execute a single action per discrete round. Each round has an evaluation phase and an execution phase.
	k-Isolated	No amoebots within distance k can be simultaneously active.
	Sequential*	At most one amoebot is active per time.
Fairness	Unfair†	Some enabled amoebot is eventually activated.
	Weakly Fair*	Every continuously enabled amoebot is eventually activated.
	Strongly Fair	Every amoebot enabled infinitely often is activated infinitely often.

Two amoebots occupying adjacent nodes are said to be *neighbors*. Although each amoebot is *anonymous*, lacking a unique identifier, we assume an amoebot can locally identify its neighbors using their port labels. In particular, we assume that amoebots A and B connected via ports p_A and p_B each know one another's labels for p_A and p_B. If A is expanded, we also assume B knows the direction A is expanded in with respect to its own local direction, and vice versa. This is sufficient for an amoebot to reconstruct which adjacent nodes are occupied by the same neighbor, but is not so strong so as to collapse the hierarchy of orientation assumptions. More details on an amoebot's anatomy are given in Section 2.1.

An amoebot's functionality is partitioned between a higher-level *application layer* and a lower-level *system layer*. Algorithms controlling an amoebot's behavior are designed from the perspective of the application layer. The system layer is responsible for an amoebot's core functions and exposes a limited programming interface of *operations* to the application layer that can be used in amoebot algorithms. The operations are defined in Section 2.2 and their organization into algorithms is described in Section 2.3. We note that future publications may abstract away from the system layer and focus on the interface to the application layer.

2.1 Amoebot Anatomy

Each amoebot has memory whose size is a model variant; the standard assumption is *constant-size* memory. An amoebot's memory consists of two parts: a persistent *public memory* that is read-writeable by the system layer but only accessible to the application layer via communication operations (see Section 2.2), and a volatile *private memory* that is inaccessible

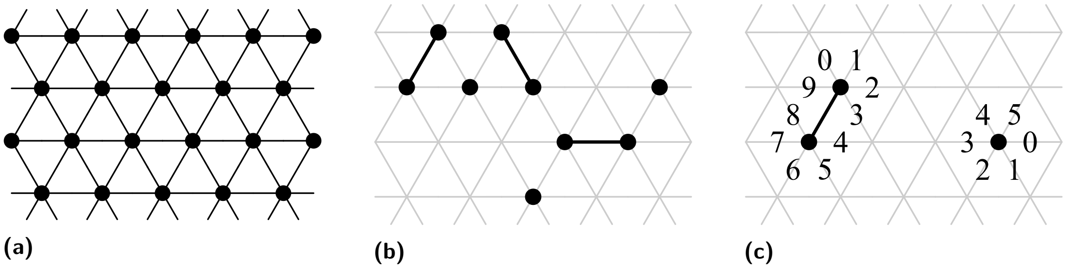


Figure 1 The Canonical Amoebot Model. (a) A section of the triangular lattice G_Δ used in the geometric variant; nodes of V are shown as black circles and edges of E are shown as black lines. (b) Expanded and contracted amoebots; G_Δ is shown in gray, and amoebots are shown as black circles. Amoebots with a black line between their nodes are expanded. (c) Two amoebots that agree on their chirality but not on their direction, using different offsets for their clockwise-increasing port labels.

to the system layer but read-writeable by the application layer. The public memory of an amoebot A contains (i) the shape of A, denoted $A.\texttt{shape} \in \{\textsc{contracted}, \textsc{expanded}\}$, ($ii$) the lock state of A, denoted $A.\texttt{lock} \in \{\textsc{true}, \textsc{false}\}$, and ($iii$) any variables used in the algorithm being run by the application layer.

Neighboring amoebots (i.e., those occupying adjacent nodes) form *connections* via their ports facing each other. An amoebot's system layer receives instantaneous feedback whenever a new connection is formed or an existing connection is broken. Communication between connected neighbors is achieved via *message passing.* To facilitate message passing communication, each of an amoebot's ports has a FIFO *outgoing message buffer* managed by the system layer that can store up to a fixed (constant) number of messages waiting to be sent to the neighbor incident to the corresponding port. If two neighbors disconnect due to some movement, their system layers immediately flush the corresponding message buffers of any pending messages. Otherwise, we assume that any pending message is sent to the connected neighbor in FIFO order in finite time. Incoming messages are processed as they are received.

2.2 Amoebot Operations

Operations provide the application layer with a programming interface for controlling the amoebot's behavior; the application layer calls operations and the system layer executes them. We assume the execution of an operation is *blocking* for the application layer; that is, the application layer can only execute one operation at a time. We summarize the communication, movement, and concurrency control operations below and in Table 2; see the full version [17] for pseudocode and execution details, including failure handling and contention resolution.

- The Connected operation checks for the presence of neighbors. Connected(p) returns true iff there is a neighbor connected via port p. Connected() snapshots current port connectivity, including whether any neighbor is connected via multiple consecutive ports.
- The Read and Write operations exchange information in public memory. Read(p, x) issues a request to read the value of a variable x in the public memory of the neighbor connected via port p while Write(p, x, x_{val}) issues a request to update its value to x_{val}. If $p = \bot$, an amoebot's own public memory is accessed instead of a neighbor's.
- An expanded amoebot can Contract into either node it occupies; a contracted amoebot can Expand into an unoccupied adjacent node. Neighboring amoebots can coordinate their movements in a *handover*, which can occur in one of two ways. A contracted amoebot A can Push an expanded neighbor B by expanding into a node occupied by B, forcing it to contract. Alternatively, an expanded amoebot B can Pull a contracted neighbor A by contracting, forcing A to expand into the neighbor it is vacating.

Table 2 Summary of operations exposed by an amoebot's system layer to its application layer.

Operation	Return Value on Success
Connected(p)	TRUE iff a neighboring amoebot is connected via port p
Connected()	$[c_0, \ldots, c_{k-1}] \in \{N_1, \ldots, N_8, \text{FALSE}\}^k$ where $c_p = N_i$ if N_i is the locally identified neighbor connected via port p and $c_p = \text{FALSE}$ otherwise
Read(p, x)	The value of x in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise
Write(p, x, x_{val})	Confirmation that the value of x was updated to x_{val} in the public memory of this amoebot if $p = \perp$ or of the neighbor incident to port p otherwise
Contract(v)	Confirmation of the contraction out of node $v \in \{\text{HEAD}, \text{TAIL}\}$
Expand(p)	Confirmation of the expansion into the node incident to port p
Pull(p)	Confirmation of the pull handover with the neighbor incident to port p
Push(p)	Confirmation of the push handover with the neighbor incident to port p
Lock()	Local identifiers of this amoebot and the neighbors that were locked
Unlock($\mathcal{L}$)	Confirmation that the amoebots of $\mathcal{L}$ were unlocked

- The Lock operation encapsulates a variant of the mutual exclusion problem where an amoebot attempts to gain exclusive control over itself and its neighbors. A Lock operation attempts to set the amoebot's and its neighbors' `lock` states from FALSE to TRUE but may fail if the amoebot or some neighbor is already locked or if there is a lock conflict. The Unlock operation releases locks obtained in a successful Lock operation.

Each operation's message passing implementation is carefully designed so that (i) any operation execution terminates – either successfully or in failure – in finite time, and (ii) at any time, there are at most a constant number of messages being sent or received between any pair of neighboring amoebots as a result of any set of operation executions. Combined with the blocking assumption, these design principles prohibit outgoing message buffer overflow and deadlocks in operation executions; see [17] for details.

2.3 Amoebot Actions, Algorithms and Executions

Following the message passing literature, we specify distributed algorithms in the amoebot model as sets of *actions* to be executed by the application layer, each of the form:

$$\langle label \rangle : \langle guard \rangle \to \langle operations \rangle$$

An action's *label* specifies its name. Its *guard* is a Boolean predicate determining whether an amoebot A can execute it based on the ports A has connections on – i.e., which nodes adjacent to A are (un)occupied – and information from the public memories of A and its neighbors. An action is *enabled* for an amoebot A if its guard is true for A, and an amoebot is *enabled* if it has at least one enabled action. An action's *operations* specify the finite sequence of operations and computation in private memory to perform if this action is executed. The control flow of this private computation may optionally include *randomization* to generate random values and *error handling* to address any operation executions resulting in failure.

Each amoebot executes its own algorithm instance independently and – as an assumption for this work – *reliably*, meaning there are no crash or Byzantine faults.[1] An amoebot is said to be *active* if its application layer is executing an action and is *idle* otherwise. An amoebot can begin executing an action if and only if it is idle; i.e., an amoebot can execute at most one action at a time. On becoming active, an amoebot A first evaluates which of its actions $\alpha_i : g_i \to ops_i$ are enabled. Since each guard g_i is based only on the connected ports of A and the public memories of A and its neighbors, each g_i can be evaluated using the Connected and Read operations. If no action is enabled, A returns to idle; otherwise, A chooses the highest-priority enabled action $\alpha_i : g_i \to ops_i$ – where action priorities are set by the algorithm – and executes the operations and private computation specified by ops_i. Recall from Section 2.2 that each operation is guaranteed to terminate (either successfully or with a failure) in finite time. Thus, since A is reliable and ops_i consists of a finite sequence of operations and finite computation, each action execution is also guaranteed to terminate in finite time after which A returns to idle. An action execution *fails* if any of its operations' executions result in a failure that is not addressed with error handling and *succeeds* otherwise.

As is standard in the distributed computing literature (see, e.g., [1]), we assume an *adversary* (or *daemon*) controls the timing of amoebot activations and the resulting action executions. The power of an adversary is determined by its *concurrency* and *fairness*. We distinguish between four concurrency variants: *sequential*, in which at most one amoebot can be active at a time; *k-isolated*, in which any set of amoebots except those occupying nodes of G_Δ within distance k can be simultaneously active; *synchronous*, in which time is discretized into "rounds" and in each round any set of amoebots can simultaneously execute one action each; and *asynchronous*, in which any amoebot can be active at any time. For synchronous concurrency, we further assume that each round is partitioned into an *evaluation phase* when all active amoebots evaluate their guards followed by an *execution phase* when all active amoebots with enabled actions execute the corresponding operations. Fairness restricts how often the adversary must activate enabled amoebots. We distinguish between three fairness variants: *strongly fair*, in which every amoebot that is enabled infinitely often is activated infinitely often; *weakly fair*, in which every continuously enabled amoebot is eventually activated; and *unfair*, in which an amoebot may never be activated unless it is the only one with an enabled action. An algorithm execution is said to *terminate* if eventually all amoebots are idle and no amoebot is enabled; note that since an amoebot can only become enabled if something changes in its neighborhood, termination is permanent. Formal notions of algorithm runtime should be defined on a per-adversary basis; e.g., a standard synchronous round would apply to synchronous adversaries, while a strongly fair sequential adversary could use rounds that complete once every amoebot has been activated at least once.

In this paper, we focus on unfair sequential and asynchronous adversaries. In the sequential setting, an active amoebot knows that all other amoebots are idle; thus, its guard evaluations must be correct since the corresponding Connected and Read operations cannot fail or have their results be outdated due to concurrent changes in the system. In the asynchronous setting, however, these issues may lead to guards being evaluated incorrectly, causing disabled actions to be executed or enabled actions to be skipped. We address these issues in two ways, justifying the formulation of algorithms in terms of actions. In Section 3, we present an algorithm whose actions are carefully designed so that their guards are always evaluated correctly under any adversary. In Section 4, we present a concurrency control framework that uses locks to ensure that guards can be evaluated correctly even in the asynchronous setting.

[1] As we discuss in Section 5, designing *fault tolerant* algorithms is an important research direction for programmable matter. We leave the formalization of different fault models under the canonical amoebot model for future work.

3 Asynchronous Hexagon Formation Without Locks

We use the *hexagon formation* problem as a concrete case study for algorithm design, pseudocode, and analysis in the canonical amoebot model. Our Hexagon-Formation algorithm (Algorithm 1) assumes geometric space, assorted orientation, and constant-size memory (Table 1) and is formulated in terms of actions as specified in Section 2.3. Our analysis establishes a general set of sufficient conditions for amoebot algorithm correctness under unfair asynchronous adversaries: (*i*) correctness under any unfair sequential adversary, (*ii*) enabled actions remaining enabled despite concurrent action executions, and (*iii*) executions of enabled actions remaining successful and unaffected by concurrent action executions. Any concurrent execution of an algorithm satisfying (*ii*) and (*iii*) can be shown to be serializable, which combined with sequential correctness establishes correctness under any unfair asynchronous adversary, the most general of all possible adversaries. Notably, we prove that our Hexagon-Formation algorithm satisfies these sufficient conditions without using locks, demonstrating that while locks are useful tools for designing correct amoebot algorithms under concurrent adversaries, they are not always necessary.

The hexagon formation problem tasks an arbitrary, connected system of initially contracted amoebots with forming a regular hexagon (or as close to one as possible, given the number of amoebots in the system). We assume that there is a *unique seed amoebot* in the system and all other amoebots are initially *idle*.[2] Following the sequential algorithm given by Derakhshandeh et al. [16, 20], the basic idea of our Hexagon-Formation algorithm is to form a hexagon by extending a spiral of amoebots counter-clockwise from the seed.

In addition to the shape variable assumed by the amoebot model, each amoebot A keeps variables $A.\texttt{state} \in \{\text{SEED}, \text{IDLE}, \text{FOLLOWER}, \text{ROOT}, \text{RETIRED}\}$, $A.\texttt{parent} \in \{\text{NULL}, 0, \ldots, 9\}$, and $A.\texttt{dir} \in \{\text{NULL}, 0, \ldots, 9\}$ in public memory. The amoebot system first self-organizes as a spanning forest rooted at the seed amoebot using their `parent` ports. Follower amoebots follow their parents until reaching the surface of retired amoebots that have already found their place in the hexagon. They then become roots, traversing the surface of retired amoebots clockwise. Once they connect to a retired amoebot's `dir` port, they also retire and set their `dir` port to the next position of the hexagon. Algorithm 1 describes Hexagon-Formation in terms of actions. We assume that if multiple actions are enabled for an amoebot, the enabled action with smallest index is executed. For conciseness and clarity, we write action guards as logical statements as opposed to their implementation with READ and CONNECTED operations. In action guards, we use $N(A)$ to denote the neighbors of amoebot A and say that an amoebot A has a *tail-child* B if B is connected to the tail of A via port $B.\texttt{parent}$.

We begin our analysis of the Hexagon-Formation algorithm by showing it is correct under any sequential adversary.[3] All omitted proofs can be found in [17].

▶ **Lemma 1.** *Any unfair sequential execution of the* Hexagon-Formation *algorithm terminates with the amoebot system forming a hexagon.*

We next consider unfair asynchronous executions, the most general of all possible adversaries. In general, asynchronous executions may cause amoebots to incorrectly evaluate their action guards. Nevertheless, in the following two lemmas, we show that Hexagon-Formation has the key property that whenever an amoebot thinks an action is enabled, it remains enabled and will execute successfully, even when other actions are executed concurrently.

[2] Note that the assumption of a unique seed amoebot immediately collapses the hierarchy of orientation assumptions since it can impose its own local orientation on the rest of the system via a simple broadcast.

[3] Although the related algorithm of Derakhshandeh et al. has already been analyzed in the sequential setting [16, 20], Hexagon-Formation must be proved correct with respect to its action formulation.

Algorithm 1 Hexagon-Formation for Amoebot A.

```
$\alpha_1 : (A.\texttt{state} \in \{\textsc{idle}, \textsc{follower}\}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\textsc{seed}, \textsc{retired}\}) \rightarrow$
    $\textsc{Write}(\bot, \texttt{parent}, \textsc{null})$.
    $\textsc{Write}(\bot, \texttt{state}, \textsc{root})$.
    $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
$\alpha_2 : (A.\texttt{state} = \textsc{idle}) \wedge (\exists B \in N(A) : B.\texttt{state} \in \{\textsc{follower}, \textsc{root}\}) \rightarrow$
    Find a port $p$ for which $\textsc{Connected}(p) = \textsc{true}$ and $\textsc{Read}(p, \texttt{state}) \in \{\textsc{follower}, \textsc{root}\}$.
    $\textsc{Write}(\bot, \texttt{parent}, p)$.
    $\textsc{Write}(\bot, \texttt{state}, \textsc{follower})$.
$\alpha_3 : (A.\texttt{shape} = \textsc{contracted}) \wedge (A.\texttt{state} = \textsc{root}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \textsc{idle})$
      $\wedge\ (\exists B \in N(A) : (B.\texttt{state} \in \{\textsc{seed}, \textsc{retired}\}) \wedge (B.\texttt{dir}$ is connected to $A)) \rightarrow$
    $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{clockwise}))$.
    $\textsc{Write}(\bot, \texttt{state}, \textsc{retired})$.
$\alpha_4 : (A.\texttt{shape} = \textsc{contracted}) \wedge (A.\texttt{state} = \textsc{root}) \wedge$ (the node adjacent to $A.\texttt{dir}$ is empty) $\rightarrow$
    $\textsc{Expand}(A.\texttt{dir})$.
$\alpha_5 : (A.\texttt{shape} = \textsc{expanded}) \wedge (A.\texttt{state} \in \{\textsc{follower}, \textsc{root}\}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \textsc{idle})$
      $\wedge\ (A$ has a tail-child $B : B.\texttt{shape} = \textsc{contracted}) \rightarrow$
    **if** $\textsc{Read}(\bot, \texttt{state}) = \textsc{root}$ **then** $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
    Find a port $p \in \textsc{TailChildren}()$ s.t. $\textsc{Read}(p, \texttt{shape}) = \textsc{contracted}$.
    Let $p'$ be the label of the tail-child's port that will be connected to $p$ after the pull handover.
    $\textsc{Write}(p, \texttt{parent}, p')$.
    $\textsc{Pull}(p)$.
$\alpha_6 : (A.\texttt{shape} = \textsc{expanded}) \wedge (A.\texttt{state} \in \{\textsc{follower}, \textsc{root}\}) \wedge (\forall B \in N(A) : B.\texttt{state} \neq \textsc{idle})$
      $\wedge\ (A$ has no tail-children$) \rightarrow$
    **if** $\textsc{Read}(\bot, \texttt{state}) = \textsc{root}$ **then** $\textsc{Write}(\bot, \texttt{dir}, \textsc{GetNextDir}(\text{counter-clockwise}))$.
    $\textsc{Contract}(\textsc{tail})$.
```

▶ **Lemma 2.** *For any asynchronous execution of the* Hexagon-Formation *algorithm, if an action α_i is enabled for an amoebot A, then α_i stays enabled for A until A executes an action.*

▶ **Lemma 3.** *For any asynchronous execution of the* Hexagon-Formation *algorithm, any execution of an enabled action is successful and unaffected by any concurrent action executions.*

Combined, Lemmas 1–3 directly imply that the Hexagon-Formation algorithm is both serializable and correct under any unfair asynchronous adversary.

▶ **Lemma 4.** *For any asynchronous execution of the* Hexagon-Formation *algorithm, there exists a sequential ordering of its action executions producing the same final configuration.*

▶ **Lemma 5.** *Any unfair asynchronous execution of the* Hexagon-Formation *algorithm terminates with the amoebot system forming a hexagon.*

Our analysis culminates in the following theorem.

▶ **Theorem 6.** *Assuming geometric space, assorted orientations, and constant-size memory, the* Hexagon-Formation *algorithm solves the hexagon formation problem under any adversary.*

We note that serializability and correctness under any asynchronous adversary (Lemmas 4–5) follow directly from Lemmas 1–3, independent of the specific details of Hexagon-Formation. Thus, these three lemmas establish a set of general sufficient conditions for amoebot algorithm correctness under asynchronous adversaries. We are hopeful that other existing amoebot algorithms can be adapted to satisfy these conditions under the new canonical model.

4 A General Framework for Concurrency Control

In the sequential setting where only one amoebot is active at a time, operation failures are necessarily the fault of the algorithm designer: e.g., attempting to Read on a disconnected port, attempting to Expand when already expanded, etc. Barring these design errors, it suffices to focus only on the correctness of the algorithm – i.e., whether the algorithm's actions always produce the desired system behavior under any sequential execution – not whether the individual actions themselves execute as intended. This is the focus of most existing amoebot works [2, 6, 7, 14, 15, 20, 21, 22, 23, 28, 31].

Our present focus is on asynchronous executions, where concurrent action executions can mutually interfere, affect outcomes, and cause failures far beyond those of simple designer negligence. Ensuring algorithm correctness in spite of concurrency thus appears to be a significant burden for the algorithm designer, especially for problems that are challenging even in the sequential setting due to the constraints of constant-size memory, assorted orientation, and strictly local interactions. What if there was a way to ensure that correct, sequential amoebot algorithms could be lifted to the asynchronous setting without sacrificing correctness? This would give the best of both worlds: the relative ease in design from the sequential setting and the correct execution in a more realistic concurrent setting.

In this section, we introduce and rigorously analyze a framework for transforming an algorithm $\mathcal{A}$ that works correctly for every sequential execution into an algorithm $\mathcal{A}'$ that works correctly for every asynchronous execution. We prove that our framework achieves this goal so long as the original algorithms satisfy certain *conventions*. These conventions limit the full generality of the amoebot model in order to provide a common structure to the algorithms. We discuss interesting open problems regarding what algorithms are compatible with this framework and whether it can be extended beyond these conventions in Section 5.

The first of these conventions requires that all actions of the given algorithm are executed successfully under a sequential adversary. For sequential executions, the *system configuration* is defined as the mapping of amoebots to the node(s) they occupy and the contents of each amoebot's public memory. Certainly, this configuration is well-defined whenever all amoebots are idle, and we call a configuration *legal* whenever the requirements of our amoebot model are met, i.e., every position is occupied by at most one amoebot, each amoebot is either contracted or expanded, its `shape` variable corresponds to its physical shape, and its `lock` variable is TRUE if and only if it has been locked in a Lock operation. Whenever we talk about a system configuration in the following, we assume that it is legal.

▶ **Convention 1.** *All actions of an amoebot algorithm should be valid, i.e., for all its actions α and all system configurations in which α is enabled for some amoebot A, the execution of α by A should be successful whenever all other amoebots are idle.*

The second convention keeps an algorithm's actions simple by controlling the order and number of operations they perform.

▶ **Convention 2.** *Each action of an amoebot algorithm should structure its operations as:*

1. *A compute phase, during which an amoebot performs a finite amount of computation in private memory and a finite sequence of Connected, Read, and Write operations.*
2. *A move phase, during which an amoebot performs at most one movement operation decided upon in the compute phase.*

In particular, no action should use Lock or Unlock operations.

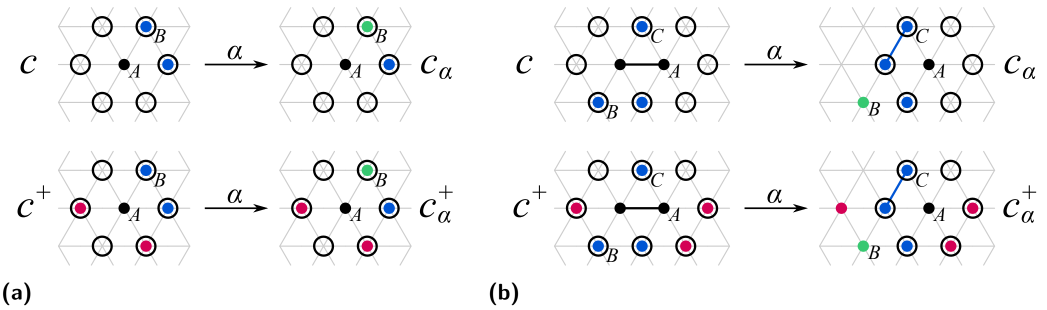


Figure 2 The monotonicity convention. In both examples, we examine the local configuration c of amoebot A, an extension c^+ of c, and the corresponding outcomes c_α and c_α^+ reached by sequential executions of α. The nodes with black circles denote the $N(\cdot)$ sets of nodes adjacent to A. Neighbors occupying the "non-extended neighborhood" $O(c)$ are shown in blue and neighbors occupying the "extended neighborhood" $O(c^+) \setminus O(c)$ are shown in pink. Monotonicity requires that the executions of α make the same updates: in (a), amoebot A updates the public memory of B, shown in green; in (b), amoebot A updates the public memory of B (green) and pulls neighbor C in a handover.

Convention 2 is similar in spirit to the *look-compute-move* paradigm used in the mobile robots literature (see, e.g., [29]), though message passing communication via READ and WRITE operations adds additional complexity in the amoebot model. Moreover, the instantaneous snapshot performed in the mobile robots' look phase is not trivially realizable by amoebots whose public memories are included in neighborhood configurations (Section 1.1).

Finally, due to our approach of using a serializability argument to show the correctness of algorithms using our framework, we need one last convention. This final convention is significantly more technical and limits the generality of the model more strictly than the first two, which we discuss further in Section 5. Consider any action $\alpha : g \to ops$ of an algorithm $\mathcal{A}$ being executed by an amoebot A. Recall that the guard g is a Boolean predicate based on the *local configuration* of A; i.e., the connected ports of A and the contents of the public memories of A and its neighbors. For a local configuration c of A, let $N(c) = (v_1, \ldots, v_k)$ be the nodes adjacent to A; note that $k = 6$ if A is contracted and $k = 8$ if A is expanded. Let $O(c) \subseteq N(c)$ be the nodes adjacent to A that are occupied by neighboring amoebots. Letting $M_\mathcal{A}$ denote the set of all possible contents of an amoebot's public memory w.r.t. algorithm $\mathcal{A}$, we can write c as a tuple $c = (c_0, c_1, \ldots, c_k) \in M_\mathcal{A} \times (M_\mathcal{A} \cup \{\text{NULL}\})^k$ where $c_0 \in M_\mathcal{A}$ is the public memory contents of A and, for $i \in \{1, \ldots, k\}$, $c_i \in M_\mathcal{A}$ is the public memory contents of the neighbor occupying node $v_i \in N(c)$ if $v_i \in O(c)$ and $c_i = \text{NULL}$ otherwise. Thus, we can express the guard g as a function $g : M_\mathcal{A} \times (M_\mathcal{A} \cup \{\text{NULL}\})^k \to \{\text{TRUE}, \text{FALSE}\}$.

A local configuration c is *consistent* if $c_i = c_j$ whenever nodes $v_i, v_j \in N(c)$ are occupied by the same expanded neighbor. Local configurations c and c' with $N(c) = N(c')$ are said to *agree on a subset* $S \subseteq N(c)$ if for all nodes $v_i \in S$, we have $c_i = c'_i$. A local configuration c^+ is an *extension* of local configuration c if c^+ is consistent and c and c^+ agree on the node A occupies and $O(c)$; intuitively, an extension c^+ has the same amoebots in the same positions with the same public memory contents as c, but may also have additional neighbors occupying nodes of $N(c) \setminus O(c)$. An extension c^+ of c is *expansion-compatible* with an execution of an action on c if any EXPAND operation by A in this execution would also succeed in c^+. An action $\alpha : g \to ops$ is *monotonic* (see Figure 2) if for any consistent local configuration c with $g(c) = \text{TRUE}$, any local configuration c_α reachable by an (isolated) execution of α on c, and any extension c^+ of c that is expansion-compatible with the execution reaching c_α and is reachable by a sequential execution of $\mathcal{A}$, $g(c^+) = \text{TRUE}$ and there exists a local configuration c_α^+ reachable by an execution of α on c^+ such that:

1. $N(c_\alpha) = N(c_\alpha^+)$; i.e., both executions of α leave A with the same set of adjacent nodes.
2. c_α and c_α^+ agree on the node A occupies and $O(c)$; i.e., both executions of α make identical updates w.r.t. A and its non-extended neighborhood.
3. c^+ and c_α^+ agree on $O(c^+) \setminus O(c)$; i.e., the execution of α on c^+ does not make any updates to the extended neighborhood of A.

▶ **Convention 3.** *All actions of an amoebot algorithm should be monotonic.*

4.1 The Concurrency Control Framework

Our *concurrency control framework* (Algorithm 2) takes as input any amoebot algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ satisfying Conventions 1–3 and produces a corresponding algorithm $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$ composed of a single action α'. The core idea of our framework is to carefully incorporate locks in α' as a wrapper around the actions of $\mathcal{A}$, ensuring that $\mathcal{A}'$ only produces outcomes in concurrent settings that $\mathcal{A}$ can produce in the sequential setting. With locks, action guards that in general can only be evaluated reliably in the sequential setting can now also be evaluated reliably in concurrent settings.

To avoid any deadlocks that locking may cause, our framework adds an *activity bit* variable $A.\texttt{act} \in \{\text{TRUE}, \text{FALSE}\}$ to the public memory of each amoebot A indicating if any changes have occurred in the memory or neighborhood of A since it last attempted to execute an action. The single action α' of $\mathcal{A}'$ has guard $g' = (A.\texttt{act} = \text{TRUE})$, ensuring that α' is only enabled for an amoebot A if changes in its memory or neighborhood may have caused some actions of $\mathcal{A}$ to become enabled. As will become clear in the presentation of the framework, WRITE and movement operations may enable actions of $\mathcal{A}$ not only for the neighbors of the acting amoebot, but also for the neighbors of those neighbors (i.e., in the 2-neighborhood of the acting amoebot). The acting amoebot cannot directly update the activity bits of amoebots in its 2-neighborhood, so it instead sets its neighbors' *awaken bits* $A.\texttt{awaken} \in \{\text{TRUE}, \text{FALSE}\}$ to indicate that they should update their neighbors' activity bits in their next action. Initially, $A.\texttt{act} = \text{TRUE}$ and $A.\texttt{awaken} = \text{FALSE}$ for all amoebots A.

Algorithm $\mathcal{A}'$ only contains one action $\alpha' : g' \to ops'$ where g' requires that an amoebot's activity bit is set to TRUE (Step 1). If α' is enabled for an amoebot A, A first attempts to LOCK itself and its neighbors (Step 2). Given that it locks successfully, there are two cases. If $A.\texttt{awaken} = \text{TRUE}$, then A must have previously been involved in the operation of some acting amoebot that changed the neighborhood of A but could not update the corresponding neighbors' activity bits (Steps 14, 17, 24, or 28). So A updates the intended activity bits to TRUE, resets $A.\texttt{awaken}$, releases its locks, and aborts (Steps 4–6). Otherwise, A obtains the necessary information to evaluate the guards of all actions in algorithm $\mathcal{A}$ (Steps 7–9). If no action from $\mathcal{A}$ is enabled for A, A sets $A.\texttt{act}$ to FALSE, releases its locks, and aborts; this disables α' for A until some future change occurs in its neighborhood (Step 10). Otherwise, A chooses any enabled action and executes its compute phase in private memory (Step 11) to determine which WRITE and movement operations, if any, it wants to perform (Step 12).

Before enacting these operations (thereby updating the system's configuration) amoebot A must be certain that no operation of α' will fail. It has already passed its first point of failure: the LOCK operation in Step 2. But α' may also fail during an EXPAND operation if it conflicts with some other concurrent expansion (Step 14). In either case, A releases any locks it obtained (if any) and aborts (Steps 3 and 15). Provided neither of these failures occur, A can now perform operations that – without locks on its neighbors – could otherwise interfere with its neighbors' actions or be difficult to undo. This begins with A setting the activity bits of all its locked neighbors to TRUE since it is about to cause activity in its neighborhood

(Step 16). It then enacts the Write operations it decided on during its computation, writing updates to its own public memory and the public memories of its neighbors. Since writes to its neighbors can change what amoebots in its 2-neighborhood see, it must also set the awaken bits of the neighbors it writes to to true (Step 17).

The remainder of the framework handles movements and releases locks. If A did not want to move or it intended to Expand – which, recall, it already did in Step 14 – it can simply release all its locks (Step 18). If A wants to contract, it must first release its locks on the neighbors it is contracting away from; it can then Contract and, once contracted, release its remaining locks (Step 20–22). If A wants to perform a Push handover, it does so and then releases all its locks (Steps 24–26). Finally, pull handovers are handled similarly to contractions: A first releases its locks on the neighbors it is disconnecting from; it can then Pull and, once contracted, release its remaining locks (Steps 28–31).

4.2 Analysis

The full technical analysis of the concurrency control framework is included in [17]. Here, we outline our argument and techniques, concluding with a statement of our main theorem.

Let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 1–3 and $\mathcal{A}'$ be the algorithm produced from $\mathcal{A}$ by our concurrency control framework (Algorithm 2). Our goal is to show that if any sequential execution of $\mathcal{A}$ terminates, then any asynchronous execution of $\mathcal{A}'$ must also terminate and will do so in a configuration that was reachable by a sequential execution of $\mathcal{A}$. This analysis is broken into two stages: analyzing $\mathcal{A}'$ under sequential executions and then leveraging a serialization argument to analyze $\mathcal{A}'$ under asynchronous executions. In each stage, we show that executions of $\mathcal{A}'$ are finite; i.e., they must terminate. Since all executions of $\mathcal{A}'$ terminate, it suffices to show that the final configurations reachable by asynchronous executions of $\mathcal{A}'$ are contained in those reachable by sequential executions of $\mathcal{A}'$ which in turn are contained in those reachable by sequential executions of $\mathcal{A}$.

The analysis of $\mathcal{A}'$ under sequential executions is relatively straightforward. We first show that every sequential execution of $\mathcal{A}'$ must terminate since otherwise there would exist an infinite sequential execution of $\mathcal{A}$, a contradiction. We then show that the added activity and awaken bits used in $\mathcal{A}'$ never cause it to terminate while there are still amoebots with actions of $\mathcal{A}$ left to perform. Together, these results imply that sequential executions of $\mathcal{A}'$ always terminate in configurations that sequential executions of $\mathcal{A}$ could also have terminated in.

The remainder of the analysis uses a serialization argument to show that asynchronous executions of $\mathcal{A}'$ always terminate in configurations that sequential executions of $\mathcal{A}'$ could also have terminated in. We model an asynchronous execution as a mapping of events (associated with action and operation executions) to ideal wall-clock times defined by an adversary. Given any asynchronous execution of $\mathcal{A}'$, we first *sanitize* it of all events associated with "irrelevant" action executions that do not affect the system configuration. We then argue that the sanitized execution remains valid (i.e., it is possible for some execution of $\mathcal{A}'$ to produce the events in the sanitized version) and that the sanitized execution changes the system configuration in exactly the same way as the original execution does.

Finally, we show that a sanitized execution can be *serialized*: there exists a sequential ordering of the action executions in the sanitized execution that produces the same final configuration as the sanitized execution. To do so, we construct a graph on the action executions in the sanitized schedule where directed edges represent causal relationships. By proving that the resulting graph is directed and acyclic, we obtain a partial order on the action executions. Since Convention 3 is satisfied, this partial order can be harnessed to obtain the desired serialization: a sequential execution of $\mathcal{A}'$ that reaches the same final configuration as the original asynchronous execution did. All together, we obtain the culminating theorem:

▶ **Theorem 7.** *Let $\mathcal{A}$ be any amoebot algorithm satisfying Conventions 1–3 and $\mathcal{A}'$ be the amoebot algorithm produced from $\mathcal{A}$ by the concurrency control framework. Let C_0 be any initial configuration for $\mathcal{A}$ and let C_0' be its extension for $\mathcal{A}'$ with $A.\texttt{act} = \text{TRUE}$ and $A.\texttt{awaken} = \text{FALSE}$ for all amoebots A. If every sequential execution of $\mathcal{A}$ starting in C_0 terminates, then every asynchronous execution of $\mathcal{A}'$ starting in C_0' terminates in a configuration that some sequential execution of $\mathcal{A}$ starting in C_0 also terminates in.*

5 Discussion and Future Work

An immediate application of the canonical amoebot model and its hierarchy of assumption variants is a systematic comparison of existing amoebot algorithms and their assumptions. For example, when comparing the two state-of-the-art amoebot algorithms for leader election using the canonical hierarchy, we find that among other problem-specific differences, Bazzi and Briones [5] assume an asynchronous adversary and common chirality while Emek et al. [28] assume a sequential adversary and assorted orientations. Such comparisons will provide valuable and comprehensive understanding of the state of amoebot literature and will facilitate clearer connections to related models of programmable matter.

Another direction would be to extend the canonical amoebot model to address *fault tolerance*. This work assumed that all amoebots are reliable, though crash faults have been previously considered in the amoebot model for specific problems [18, 27]. Faulty amoebot behavior is especially challenging for lock-based concurrency control mechanisms which are prone to deadlock in the presence of crash faults. Additional modeling efforts will be needed to introduce a stable family of fault assumptions.

Finally, further study is needed on the design of algorithms for concurrent settings. The amoebot model's inclusion of memory, communication, and movement exacerbates issues of concurrency, ranging from operating based on stale information to conflicts of movement. Our analysis of the Hexagon-Formation algorithm produced one set of algorithm-agnostic invariants that yield correct asynchronous behavior without the use of locks (Lemmas 1–3) while our concurrency control framework gives another set of sufficient conditions for obtaining correct behavior under an asynchronous adversary when using locks (Conventions 1–3).

Of the three conventions used by the concurrency control framework, monotonicity (Convention 3) is the most restrictive and technically difficult to verify. The serializability argument relies on it to show that when an action execution is removed from its timing in an asynchronous schedule into the future where it is not concurrent with any other execution, it makes exactly the same changes to the system configuration that it did originally, regardless of any new amoebots that may have moved into its neighborhood in the meantime. In that light, it is easy to see that *stationary* algorithms that do not use movement trivially satisfy monotonicity. These include many of the existing algorithms for leader election [5, 15, 23, 25, 31] and the recent algorithm for energy distribution [18]. However, many interesting collective behaviors for programmable matter require movement, and it remains an open problem to identify if any of these algorithms satisfy monotonicity.

We emphasize that the monotonicity convention is not simply a technicality of our approach but rather a general phenomenon for asynchronous amoebot systems. Imagine a cycle alternating between contracted amoebots and empty positions and an asynchronous execution where all amoebots, having no neighbors, expand concurrently. This forms a cycle of expanded amoebots. However, any serialization of these expansions would result in at least one amoebot seeing an already expanded neighbor at the start of its action execution, which may prohibit its expansion and stop the system from reaching the original outcome (an expanded cycle).

This discussion highlights two critical open questions. Do there exist algorithms that are not correct under an asynchronous adversary but are compatible with our concurrency control framework? Are there other, less restrictive sufficient conditions for correctness in spite of asynchrony? We are hopeful that our approaches to concurrent algorithm design combined with answers to these open problems will advance the analysis of existing and future algorithms for programmable matter in the concurrent setting.

References

1 Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*, volume 8 of *Synthesis Lectures on Distributed Computing Theory*. Morgan & Claypool Publishers, 2019.

2 Marta Andrés Arroyo, Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A stochastic approach to shortcut bridging in programmable matter. *Natural Computing*, 17(4):723–741, 2018.

3 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

4 Eduardo Mesa Barrameda, Shantanu Das, and Nicola Santoro. Deployment of asynchronous robotic sensors in unknown orthogonal environments. In *Algorithmic Aspects of Wireless Sensor Networks*, ALGOSENSORS 2008, pages 125–140, 2008.

5 Rida A. Bazzi and Joseph L. Briones. Stationary and deterministic leader election in self-organizing particle systems. In *Stabilization, Safety, and Security of Distributed Systems*, SSS 2019, pages 22–37, 2019.

6 Sarah Cannon, Joshua J. Daymude, Cem Gokmen, Dana Randall, and Andréa W. Richa. A local stochastic algorithm for separation in heterogeneous self-organizing particle systems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX/RANDOM 2019, pages 54:1–54:22, 2019.

7 Sarah Cannon, Joshua J. Daymude, Dana Randall, and Andréa W. Richa. A Markov chain algorithm for compression in self-organizing particle systems. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC 2016, pages 279–288, 2016.

8 Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In *DNA Computing and Molecular Programming*, DNA 2018, pages 155–172, 2018.

9 Gregory S. Chirikjian. Kinematics of a metamorphic robotic system. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, volume 1 of *ICRA 1994*, pages 449–455, 1994.

10 Gianlorenzo D'Angelo, Mattia D'Emidio, Shantanu Das, Alfredo Navarra, and Giuseppe Prencipe. Leader election and compaction for asynchronous silent programmable matter. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS 2020, pages 276–284, 2020.

11 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. The power of lights: Synchronizing asynchronous robots using visible bits. In *IEEE 32nd International Conference on Distributed Computing Systems*, ICDCS 2012, pages 506–515, 2012.

12 Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. Autonomous mobile robots with lights. *Theoretical Computer Science*, 609:171–184, 2016.

13 Joshua J. Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.

14 Joshua J. Daymude, Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Christian Scheideler, and Andréa W. Richa. Convex hull formation for programmable matter. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN 2020, pages 2:1–2:10, 2020.

15 Joshua J. Daymude, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Improved leader election for self-organizing programmable matter. In *Algorithms for Sensor Systems*, ALGOSENSORS 2017, pages 127–140, 2017.

16 Joshua J. Daymude, Kristian Hinnenthal, Andréa W. Richa, and Christian Scheideler. Computing by programmable particles. In *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 615–681. Springer, 2019.

17 Joshua J. Daymude, Andréa W. Richa, and Christian Scheideler. The canonical amoebot model: Algorithms and concurrency control. Full version available online at `arXiv:2105.02420`, 2021.

18 Joshua J. Daymude, Andréa W. Richa, and Jamison W. Weber. Bio-inspired energy distribution for programmable matter. In *International Conference on Distributed Computing and Networking 2021*, ICDCN 2021, pages 86–95, 2021.

19 Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: Amoebot - a new model for programmable matter. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2014, pages 220–222, 2014.

20 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, NANOCOM 2015, pages 21:1–21:2, 2015.

21 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA 2016, pages 289–299, 2016.

22 Zahra Derakhshandeh, Robert Gmyr, Andréa W. Richa, Christian Scheideler, and Thim Strothmann. Universal coating for programmable matter. *Theoretical Computer Science*, 671:56–68, 2017.

23 Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida A. Bazzi, Andréa W. Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In *DNA Computing and Molecular Programming*, DNA 2015, pages 117–132, 2015.

24 Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Mobile RAM and shape formation by programmable particles. In *Euro-Par 2020: Parallel Processing*, pages 343–358, 2020.

25 Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, 33(1):69–101, 2020.

26 Giuseppe Antonio Di Luna, Paola Flocchini, Sruti Gan Chaudhuri, Federico Poloni, Nicola Santoro, and Giovanni Viglietta. Mutual visibility by luminous robots without collisions. *Information and Computation*, 254(3):392–418, 2017.

27 Giuseppe Antonio Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Line recovery by programmable particles. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN 2018, pages 4:1–4:10, 2018.

28 Yuval Emek, Shay Kutten, Ron Lavi, and William K. Moses Jr. Deterministic leader election in programmable matter. In *46th International Colloquium on Automata, Languages, and Programming*, ICALP 2019, pages 140:1–140:14, 2019.

29 Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro, editors. *Distributed Computing by Mobile Entities*. Springer International Publishing, Switzerland, 2019.

30 Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Masafumi Yamashita. Rendezvous with constant memory. *Theoretical Computer Science*, 621:57–72, 2016.

31 Nicolas Gastineau, Wahabou Abdou, Nader Mbarek, and Olivier Togni. Distributed leader election and computation of local identifiers for programmable matter. In *Algorithms for Sensor Systems*, ALGOSENSORS 2018, pages 159–179, 2019.

32 Lindsey Hines, Kirstin Petersen, Guo Zhan Lum, and Metin Sitti. Soft actuators for small-scale robotics. *Advanced Materials*, 29(13):1603483, 2017.

33 Sam Kriegman, Douglas Blackiston, Michael Levin, and Josh Bongard. A scalable pipeline for designing reconfigurable organisms. *Proceedings of the National Academy of Sciences*, 117(4):1853–1859, 2020.

34 Albert Tianxiang Liu, Jing Fan Yang, Lexy N. LeMar, Ge Zhang, Ana Pervan, Todd D. Murphey, and Michael S. Strano. Autoperforation of two-dimensional materials to generate colloidal state machines capable of locomotion. *Faraday Discussions*, 2021.

35 Othon Michail and Paul G. Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29:207–237, 2016.

36 Matthew J. Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.

37 Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42:1619–1633, 2018.

38 Tommaso Toffoli and Norman Margolus. Programmable matter: Concepts and realization. *Physica D: Nonlinear Phenomena*, 47(1):263–272, 1991.

39 Damien Woods, Ho-Lin Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science*, ITCS 2013, pages 353–354, 2013.

40 Hui Xie, Mengmeng Sun, Xinjian Fan, Zhihua Lin, Weinan Chen, Lei Wang, Lixin Dong, and Qiang He. Reconfigurable magnetic microrobot swarm: Multimode transformation, locomotion, and manipulation. *Science Robotics*, 4(28):eaav8006, 2019.

41 Jing Fan Yang, Pingwei Liu, Volodymyr B. Koman, Albert Tianxiang Liu, and Michael S. Strano. Synthetic Cells: Colloidal-sized state machines. In Shawn M. Walsh and Michael S. Strano, editors, *Robotic Systems and Autonomous Platforms*, Woodhead Publishing in Materials, pages 361–386. Woodhead Publishing, 2019.

A Appendix: Concurrency Control Framework Pseudocode

Algorithm 2 Concurrency Control Framework for Amoebot A.

```
Input: Algorithm $\mathcal{A} = \{[\alpha_i : g_i \to ops_i] : i \in \{1, \ldots, m\}\}$ satisfying Conventions 1–3.
Set $g' \leftarrow (A.\texttt{act} = \textsc{true})$ and $ops' \leftarrow$ "Do:
    try: Set $\mathcal{L} \leftarrow \textsc{Lock}()$ to attempt to lock $A$ and its neighbors.
    catch lock-failure do abort.
    if $A.\texttt{awaken} = \textsc{true}$ then
        for all amoebots $B \in \mathcal{L}$ do Write $B.\texttt{act} \leftarrow \textsc{true}$.
        Write $A.\texttt{awaken} \leftarrow \textsc{false}$, $\textsc{Unlock}(\mathcal{L})$, and abort.
    for all actions $[\alpha_i : g_i \to ops_i] \in \mathcal{A}$ do
        Perform Connected and Read operations to evaluate guard $g_i$ w.r.t. $\mathcal{L}$.
        Evaluate $g_i$ in private memory to determine if $\alpha_i$ is enabled.
    if no action is enabled then Write $A.\texttt{act} \leftarrow \textsc{false}$, $\textsc{Unlock}(\mathcal{L})$, and abort.
    Choose an enabled action $\alpha_i \in \mathcal{A}$ and perform its compute phase in private memory.
    Let $W_i$ be the set of Write operations and $M_i$ be the movement operation in $ops_i$ based on
        its compute phase; set $M_i \leftarrow \textsc{null}$ if there is none.
    if $M_i$ is Expand (say, from node $u$ into node $v$) then
        try: Perform the Expand operation and Write $A.\texttt{awaken} \leftarrow \textsc{true}$.
        catch expand-failure do $\textsc{Unlock}(\mathcal{L})$ and abort.
    for all amoebots $B \in \mathcal{L}$ do Write $B.\texttt{act} \leftarrow \textsc{true}$.
    for all $(B.x \leftarrow x_{val}) \in W_i$ do Write $B.x \leftarrow x_{val}$ and Write $B.\texttt{awaken} \leftarrow \textsc{true}$.
    if $M_i$ is null or Expand then Unlock each amoebot in $\mathcal{L}$.
    else if $M_i$ is Contract (say, from nodes $u, v$ into node $u$) then
        Unlock each amoebot in $\mathcal{L}$ that is adjacent to node $v$ but not to node $u$.
        Perform the Contract operation.
        Unlock each remaining amoebot in $\mathcal{L}$.
    else if $M_i$ is Push (say, $A$ is pushing $B$) then
        Write $A.\texttt{awaken} \leftarrow \textsc{true}$ and $B.\texttt{awaken} \leftarrow \textsc{true}$.
        Perform the Push operation.
        $\textsc{Unlock}(\mathcal{L})$.
    else if $M_i$ is Pull (say, $A$ in nodes $u, v$ is pulling $B$ into node $v$) then
        Write $B.\texttt{awaken} \leftarrow \textsc{true}$.
        Unlock each amoebot in $\mathcal{L}$ (except $B$) that is adjacent to node $v$ but not to node $u$.
        Perform the Pull operation.
        Unlock each remaining amoebot in $\mathcal{L}$."
return $\mathcal{A}' = \{[\alpha' : g' \to ops']\}$.
```

Improved Weighted Additive Spanners

Michael Elkin
Ben-Gurion University of the Negev, Beer Sheva, Israel

Yuval Gitlitz
Ben-Gurion University of the Negev, Beer Sheva, Israel

Ofer Neiman
Ben-Gurion University of the Negev, Beer Sheva, Israel

Abstract

Graph spanners and emulators are sparse structures that approximately preserve distances of the original graph. While there has been an extensive amount of work on additive spanners, so far little attention was given to weighted graphs. Only very recently [3] extended the classical +2 (respectively, +4) spanners for unweighted graphs of size $O(n^{3/2})$ (resp., $O(n^{7/5})$) to the weighted setting, where the additive error is $+2W$ (resp., $+4W$). This means that for every pair u, v, the additive stretch is at most $+2W_{u,v}$, where $W_{u,v}$ is the maximal edge weight on the shortest $u-v$ path (weights are normalized so that the minimum edge weight is 1). In addition, [3] showed a randomized algorithm yielding a $+8W_{max}$ spanner of size $O(n^{4/3})$, here W_{max} is the maximum edge weight in the entire graph.

In this work we improve the latter result by devising a simple deterministic algorithm for a $+(6+\varepsilon)W$ spanner for weighted graphs with size $O(n^{4/3})$ (for any constant $\varepsilon > 0$), thus nearly matching the classical +6 spanner of size $O(n^{4/3})$ for unweighted graphs. Furthermore, we show a $+(2+\varepsilon)W$ subsetwise spanner of size $O(n \cdot \sqrt{|S|})$, improving the $+4W_{max}$ result of [3] (that had the same size). We also show a simple randomized algorithm for a $+4W$ emulator of size $\tilde{O}(n^{4/3})$.

In addition, we show that our technique is applicable for very sparse additive spanners, that have linear size. It is known that such spanners must suffer polynomially large stretch. For weighted graphs, we use a variant of our simple deterministic algorithm that yields a linear size $+\tilde{O}(\sqrt{n} \cdot W)$ spanner, and we also obtain a tradeoff between size and stretch.

Finally, generalizing the technique of [12] for unweighted graphs, we devise an efficient randomized algorithm producing a $+2W$ spanner for weighted graphs of size $\tilde{O}(n^{3/2})$ in $\tilde{O}(n^2)$ time.

2012 ACM Subject Classification Theory of computation → Sparsification and spanners

Keywords and phrases Graph theory, Pure additive spanners

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.21

Related Version *Previous Version*: `https://arxiv.org/abs/2008.09877`

Funding *Michael Elkin*: ISF grant 2344/19
Yuval Gitlitz: Partially supported by the Lynn and William Frankel Center for Computer Sciences and ISF grant 1817/17.
Ofer Neiman: ISF grant 1817/17

1 Introduction

Let $G = (V, E, w)$ be a weighted undirected graph on n vertices. Denote by $d_G(u, v)$ the distance between $u, v \in V$ in the graph G. A graph $H = (V, E', w')$ is an (α, β)-*spanner* of G if it is a subgraph of G and for every $u, v \in V$,

$$d_H(u, v) \le \alpha \cdot d_G(u, v) + \beta.$$

For an *emulator* H, we drop the subgraph requirement (that is, we allow H to have edges that are not present in G, while still maintaining $d_H(u, v) \ge d_G(u, v)$ for all $u, v \in V$).

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 21; pp. 21:1–21:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Spanners were introduced in the 80's by [20], and have been extensively studied ever since. One of the key objectives in this field is to understand the tradeoff between the stretch of a spanner and its size (number of edges). For purely multiplicative spanners (with $\beta = 0$), an answer was quickly given: for any integer $k \geq 1$, [5] showed that a greedy algorithm provides a $(2k-1, 0)$-spanner with size $O(n^{1+1/k})$. This bound is tight assuming Erdős' girth conjecture.

In this paper we focus on purely additive spanners, where $\alpha = 1$, which we denote by $+\beta$ spanners. Almost all of the previous work on purely additive spanners was done for unweighted graphs. The first purely additive spanner was a $+2$ spanner of size $O(n^{1.5})$ [4, 18], which was followed by a $+6$ spanner of size $O(n^{4/3})$ [6, 19], and a $+4$ spanner of size $\tilde{O}(n^{7/5})$ [11, 7]. A result of [1] showed that any purely additive spanner with $O(n^{4/3-\delta})$ edges, for constant $\delta > 0$, must have a polynomial stretch β. On the other hand, several works [22, 11, 9, 8] obtained sparser spanners with polynomial stretch. The state-of-the-art result of [8] has near-linear size and stretch $\tilde{O}(n^{3/7})$.

In [18] the notion of *near-additive* spanners for unweighted graphs was introduced, where $\alpha = 1 + \varepsilon$ for some small $\varepsilon > 0$. They showed $(1+\varepsilon, \beta)$-spanners of size $O(\beta \cdot n^{1+1/k})$ with $\beta = O(\frac{\log k}{\varepsilon})^{\log k}$. Many following works [13, 14, 23, 22, 2, 17] improved several aspects of these spanners, but up to the β factor in the size, this is still the state-of-the-art. Providing some evidence to its tightness, [2] showed that such spanners must have $\beta = \Omega(\frac{1}{\varepsilon \cdot \log k})^{\log k}$.

Since many applications of spanners stem from weighted graphs (in particular some distributed applications, such as asynchronous protocol design [21], compact routing tables [24, 16]. For more see [3] and the references therein), it is only natural to study additive spanners in that setting. Assume the weights are normalized so that the minimum edge weight is 1. We distinguish between two types of additive spanners; in the first one the additive stretch is $+c \cdot W_{\max}$, where $W_{\max}$ is the weight of heaviest edge in the graph, and c is usually some constant. A more desirable type of additive stretch is denoted by $+c \cdot W$, which means that for every $u, v \in V$,

$$d_H(u,v) \leq d_G(u,v) + c \cdot W_{u,v},$$

where $W_{u,v}$ is the heaviest edge in the shortest path between u, v in G (if there are multiple shortest paths, pick the one with the minimal heaviest edge). This estimation is not only stronger, but also handles nicely the multiplicative perspective of the spanner: a $+c \cdot W$ spanner is also a $(c+1, 0)$ spanner (while a $+W_{max}$ approximation can have unbounded multiplicative stretch).

The first adaptation of (near)-additive spanners to the weighted setting was given in [15], where we showed near-additive spanners and emulators with essentially the same stretch and size as the state-of-the-art results for unweighted graphs, while β is multiplied by W (the maximal edge weight on the corresponding path). In addition, a construction of an additive $+2W$ spanner of size $\tilde{O}(n^{3/2})$ can be inferred from [15].[1] Ahmed et al. [3] recently gave a comprehensive study of weighted additive spanners. Among other results, they showed a $+2W_{\max}$ spanner of size $O(n^{1.5})$, a $+4W$ spanner of size $O(n^{7/5})$, [2] and a $+8W_{\max}$ spanner of size $O(n^{4/3})$. Given a set $S \subseteq V$, they showed a $+4W_{max}$ subsetwise spanner of size $O(n \cdot \sqrt{|S|})$ (that has approximation guarantee only for pairs in $S \times S$). While the former two results match the state-of-the-art unweighted bounds, the latter two leave room for improvement. Indeed, [3] pose as an open problem whether a $+6W_{\max}$ spanner of size $O(n^{4/3})$ can be achieved.

[1] The notation $\tilde{O}(\cdot)$ hides polylogarithmic factors.

[2] In their paper the spanner is claimed to be $+4W_{\max}$ but a tighter analysis shows it is actually a $+4W$.

1.1 Our results

In this work we improve the bounds of [3] both quantitatively and qualitatively. For any constant $\varepsilon > 0$, we show a simple deterministic construction of a $+(6+\varepsilon)W$ spanner of size $O(n^{4/3})$.[3] Thus, the additive stretch of our spanner is arbitrarily close to $6W$, while having the superior dependence on the largest edge weight on the shortest $u-v$ path, rather than the global maximum weight. Furthermore, our algorithm is a simple greedy algorithm, in contrast to the more involved 2-stages randomized algorithm of [3].

We show the versatility of our techniques by applying them to the subsetwise setting. Given a set $S \subseteq V$, for any constant $\varepsilon > 0$, we obtain a $(2+\varepsilon)\cdot W$ subsetwise spanner of size $O(n \cdot \sqrt{|S|})$, again improving [3] both in the stretch and in the dependence on maximal edge weight.

A slight variant of our simple greedy algorithm works in the setting of sparse spanners with polynomial additive stretch, also for weighted graphs. This is in contrast to essentially all previous algorithms for very sparse pure additive spanners, that were rather involved. In particular, we obtain a linear size $+\tilde{O}(\sqrt{n})\cdot W$ spanner, and more generally, for any $0 \le \varepsilon \le 1$, a $+O(n^{\frac{1-\varepsilon}{2}}\log n)W$ spanner of size $O(n^{1+\varepsilon})$. While this result does not match the state-of-the-art for unweighted graphs, we believe it is interesting to have such spanners in the weighted setting, and we find the simplicity of the algorithm appealing.

In addition, we show a simple randomized algorithm that produces a $+4W$ emulator of size $\tilde{O}(n^{4/3})$. This corresponds to the $+4$ emulator of size $O(n^{4/3})$ for unweighted graphs [4, 18].

Finally, bearing the mind the applications of such spanners to efficiently computing shortest paths, we devise an efficient $\tilde{O}(n^2)$ time algorithm for a $+(2+\varepsilon)W$ spanner of size $\tilde{O}(n^{3/2})$ (the previous best running time was $\tilde{O}(n^{2.5})$ [15]). This result builds on the [12] $+2$ spanner for unweighted graphs.

1.2 Overview of our construction and analysis

Our algorithms for the $(6+\varepsilon)\cdot W$ spanner and the $(2+\varepsilon)\cdot W$ subsetwise spanner follow a common approach. We adapt the algorithm of [19], who showed a simple $+6$ spanner for unweighted graphs, to the weighted setting. Both [19] and the path-buying construction of [6] iteratively add paths to the spanner H, and argue that for each new edge in a path that is added to H, there is some progress for many pairs of vertices. Specifically, assume that for some $u, v \in V$ we have for a constant c that

$$d_H(u,v) \le d_G(u,v) + c\ , \tag{1}$$

where H is the current spanner we maintain. For unweighted graphs, if we make progress and improve the distance in H between u, v, it will be by at least 1. Thus, once we obtain (1), the distance between u, v can be improved at most c more times. This nice attribute does not apply to weighted graphs, since there the distance between u, v can be improved only by a tiny amount.

In our algorithm, we first add the t-lightest edges incident on every vertex (the value of t depends on the required sparsity), and then greedily add shortest paths between vertices whose stretch is too large, ordered by their W. To overcome the issue of tiny improvements, our notion of progress depends on the weights. That is, when adding paths to the spanner, we will show that many pairs improve their distance by at least $\Omega(\varepsilon \cdot W)$. Note that W is in

[3] For arbitrary $\varepsilon > 0$, the size of our spanner is $O(n^{4/3}/\varepsilon)$.

fact a function (the maximum edge weight in the current path), so some care is required to ensure sufficient progress is made for many other pairs (that can have either a smaller or a larger W). Now, if the current distance in H between $u, v \in V$ is

$$d_H(u,v) \leq d_G(u,v) + c \cdot W,$$

then the distance between u, v can be improved at most $O(\frac{c}{\varepsilon})$ more times. This number translates directly to the size of the spanner, and also affects the stretch.

The previous constructions of (near) linear-size additive spanners with polynomial stretch, such as [10, 6, 9, 8], used rather complicated constructions and analysis, based on distance preservers, path-buying, and involved clustering. In this work we show for the first time that a simple greedy algorithm, augmented by a multiplicative spanner, can also provide such a linear-size spanner. Moreover, our algorithm provides a spanner even in the weighted setting. The analysis of this algorithm is nontrivial, and uses a novel labeling scheme of the graph vertices. The idea is that each of the greedily added paths must have labeled a lot of new vertices, else we could have used the existing t-lightest edges, combined with the multiplicative spanner and the previously added paths, to obtain a sufficiently low stretch alternative path. We then conclude that the number of added paths is bounded, which is then used to bound the number of edges added to the spanner in all these paths, by an argument based on low intersections between shortest paths.

1.3 Organization

After reviewing a few preliminary results in Section 2, we show our $+(6+\varepsilon) \cdot W$ spanner in Section 3, and the linear size spanner with polynomial stretch for weighted graphs in Section 5. The $+2W$ spanner with $\tilde{O}(n^2)$ construction time is shown in Section 6. Our $+(2+\varepsilon) \cdot W$ subsetwise spanner is in Section 4, and the $+4W$ emulator in Section 7.

2 Preliminaries

Let $G = (V, E, w)$ be a weighted undirected graph, with nonnegative weights $w : E \to \mathbb{R}_+$, and fix a parameter $\varepsilon > 0$. Denote by $P_{u,v}$ the shortest path between vertices $u, v \in V$, breaking ties consistently (say by id's), so that every sub-path of a shortest path is also a shortest path and two shortest paths have at most one intersecting subpath. Let $W_{u,v}$ denote the weight of the heaviest edge in $P_{u,v}$. For a positive integer t, a *t-light initialization* of G is a subgraph $H = (V, E', w)$ that contains, for each $u \in V$, the lightest t edges incident on u (or all of them, if $\deg(u) \leq t$), breaking ties arbitrarily. For $u \in V$, we say that v is a *t-light neighbor* of u if the edge $\{u, v\}$ is among the t lightest edges incident on u.

The following lemma was shown in [3, Theorem 5].

▶ **Lemma 1** ([3]). *Let $G = (V, E, w)$ be an undirected weighted graph, and H a t-light initialization of G. If $P_{u,v}$ is some shortest path in G that is missing ℓ edges in H, then there is a set of vertices $S \subseteq V$ such that:*

1. $|S| = \Omega(t\ell)$.
2. *For each vertex $a \in S$ there exists a vertex $b \in P_{u,v}$ s.t. a is a t-light neighbor of b, with edge weight $w(a, b) \leq W_{u,v}$. In other words, all the vertices in S are connected to $P_{u,v}$ using edges lighter than $W_{u,v}$.*

(The fact that *light* edges are connecting S to $P_{u,v}$ did not appear explicitly in [3], but it follows directly from their proof.)

We will also use the construction of the greedy multiplicative spanners [5].

▶ **Lemma 2** ([5]). *Let $G = (V, E, w)$ be an undirected weighted graph, and fix a parameter $k \geq 1$. There exists a $(2k - 1, 0)$-spanner of size $O(n^{1+1/k})$.*

The following standard lemma asserts that sampling a random set S of vertices with the appropriate density, will guarantee with high probability (w.h.p.) that for every $u \in V$: either all of its neighbors are in a t-light initialization, or u has a light neighbor in S.

▶ **Lemma 3.** *Let $G = (V, E, w)$ be an undirected weighted graph and let H be a $(2n^{\varepsilon} \ln n)$-light initialization of G for some $0 \leq \varepsilon \leq 1$. Let $S \subseteq V$ be a random set, created by sampling each vertex independently with probability $\frac{1}{n^{\varepsilon}}$. Then with probability at least $1 - 1/n$, for every vertex u having at least $2n^{\varepsilon} \ln n$ neighbors in G, there exists $y \in S$ s.t. y is a $(2n^{\varepsilon} \ln n)$-light neighbor of u.*

Proof. Let U be the set of vertices with degree at least $2n^{\varepsilon} \ln n$ in G. Fix $u \in U$, and denote by X_u the event that there exists $y \in S$ which is a $(2n^{\varepsilon} \ln n)$-light neighbor of u. Every vertex is sampled to S independently with probability $\frac{1}{n^{\varepsilon}}$, hence

$$\Pr[\bar{X}_u] = \left(1 - \frac{1}{n^{\varepsilon}}\right)^{2n^{\varepsilon} \ln n} \leq (1/e)^{2 \ln n} = (1/n)^2.$$

Let X be the event that for every $u \in U$, the event X_u occur. By the union bound,

$$\Pr[\bar{X}] \leq \sum_{u \in U} \Pr[\bar{X}_u] \leq |U|/n^2 \leq 1/n.$$ ◀

3 A $+(6 + \varepsilon)W$ spanner

In this section we present our $+(6 + \varepsilon)W$ spanner which is an adaptation of the construction of [19] for weighted graphs.

Construction

Our algorithm for a $+(6 + \varepsilon)W$ spanner works as follows. Initially, H is set as a $n^{1/3}$-light initialization of G. Next, sort all the pairs $u, v \in V$: first according to $W_{u,v}$, and then by $d_G(u, v)$ (from small to large), breaking ties arbitrarily. Then, go over all pairs in this order; when considering u, v, we add $P_{u,v}$ to H if

$$d_H(u, v) > d_G(u, v) + (6 + \varepsilon)W_{u,v}. \tag{2}$$

Analysis

Our main technical lemma below asserts that by adding a shortest path to H, we get for many pairs of the path's neighbors: 1) a good initial guarantee, and also 2) sufficiently improve their distance in H.

▶ **Lemma 4.** *Let $u, v \in V$ be two vertices for which the path $P_{u,v}$ was added to H, and take any $x \in P_{u,v}$. Let $a, b, c \in V$ be different $n^{1/3}$-light neighbors of u, x, v, respectively, with edge weights at most $W_{u,v}$. Denote by H_0 the spanner just before $P_{u,v}$ was added and by H_1 the spanner right after the path was added. Then both of the following hold.*

1. $d_{H_1}(a, b) \leq d_G(a, b) + 4W_{u,v}$ ***and*** $d_{H_1}(b, c) \leq d_G(b, c) + 4W_{u,v}$.
2. $d_{H_1}(a, b) \leq d_{H_0}(a, b) - \frac{\varepsilon}{2}W_{u,v}$ ***or*** $d_{H_1}(b, c) \leq d_{H_0}(b, c) - \frac{\varepsilon}{2}W_{u,v}$.

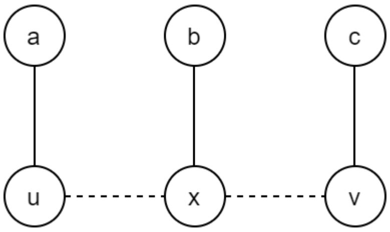


Figure 1 An illustration for Lemma 4. The dotted line is $P_{u,v}$, and the edges $\{a,u\},\{b,x\},\{c,v\}$ are all light. It is possible that $u = x$ or $v = x$.

Proof. Fix $P_{u,v}$ and a, b, c as defined in the Lemma, see also Figure 1. We begin by proving the first item, using the triangle inequality and the fact that the three edges $\{a,u\},\{b,x\},\{c,v\}$ all appear in H_1 (since they are $n^{1/3}$-light), and have weight at most $W_{u,v}$.

$$\begin{aligned} d_{H_1}(a,b) &\leq d_{H_1}(a,u) + d_{H_1}(u,x) + d_{H_1}(x,b) \\ &= w(a,u) + d_G(u,x) + w(x,b) \\ &\leq w(a,u) + d_G(u,a) + d_G(a,b) + d_G(b,x) + w(x,b) \\ &\leq d_G(a,b) + 4W_{u,v}. \end{aligned} \tag{3}$$

The bound on $d_{H_1}(b,c)$ follows in a symmetric manner, which concludes the proof of the first item. Seeking contradiction, assume that the second item does not hold. This suggests that

$$d_{H_0}(a,b) < d_{H_1}(a,b) + \frac{\varepsilon}{2}W_{u,v} \overset{(3)}{\leq} d_G(u,x) + (2+\frac{\varepsilon}{2})W_{u,v} \ ,$$

and also

$$d_{H_0}(b,c) < d_{H_1}(b,c) + \frac{\varepsilon}{2}W_{u,v} \leq d_G(x,v) + (2+\frac{\varepsilon}{2})W_{u,v} \ .$$

So we have that

$$\begin{aligned} d_{H_0}(u,v) &\leq d_{H_0}(u,a) + d_{H_0}(a,b) + d_{H_0}(b,c) + d_{H_0}(c,v) \\ &< w(u,a) + d_G(u,x) + (2+\frac{\varepsilon}{2})W_{u,v} + d_G(x,v) + (2+\frac{\varepsilon}{2})W_{u,v} + w(c,v) \\ &\leq d_G(u,v) + (6+\varepsilon)W_{u,v}, \end{aligned}$$

which is a contradiction to (2), since we assumed that the path $P_{u,v}$ was added to the spanner. ◀

▶ **Theorem 5.** *For every undirected weighted graph $G = (V, E, w)$ and $\varepsilon > 0$, there exists a deterministic polynomial time algorithm that produces a $+(6+\varepsilon)W$ spanner of size $O(\frac{1}{\varepsilon} \cdot n^{4/3})$.*

Proof. Our construction algorithm adds a shortest path between pairs whose stretch is larger than $+(6+\varepsilon)W$, so we trivially get a $+(6+\varepsilon)W$ spanner (the running time can be easily checked to be polynomial in n). Thus, we only need to bound the number of edges. Starting with the $n^{1/3}$-light initialization introduces at most $n^{4/3}$ edges to the spanner, so it remains to bound the number of edges added by adding the shortest paths.

Let $u, v \in V$ be two vertices for which the path $P_{u,v}$ was added to the spanner. Consider the time in which this path was added, let H_0 be the spanner just before the addition of $P_{u,v}$, and H_1 after the addition. We say that a pair of vertices $a, b \in V$ is *set-off* at this time, if it is the first time that $d_{H_1}(a,b) \le d_G(a,b) + 4W_{u,v}$, and it is *improved* if $d_{H_1}(a,b) \le d_{H_0}(a,b) - \frac{\varepsilon}{2}W_{u,v}$. The main observation is that once a pair is set-off, it can be improved at most $O(\frac{1}{\varepsilon})$ times. To see this, note that after the set-off we have $d_H(a,b) - d_G(a,b) \le 4W_{u,v}$, and recall that we ordered the pairs by their maximal weight $W_{u,v}$, so any future improvement will be at least by $\frac{\varepsilon}{2}W_{u,v}$. Since at the end we must have $d_H(a,b) \ge d_G(a,b)$, there can be at most $O(\frac{1}{\varepsilon})$ improvements.

We will show that if ℓ edges of $P_{u,v}$ are missing in H_0, then at least $\Omega(\ell \cdot n^{2/3})$ pairs are either set-off or improved. Fix any $x \in P_{u,v}$, and let $a,b,c \in V$ be different $n^{1/3}$-light neighbors of u,x,v, respectively, connected by edges of weight at most $W_{u,v}$. Apply Lemma 2 on u,v,x and a,b,c. We get that both pairs (a,b) and (b,c) are set-off (if they haven't before), and at least one of them is improved.

The final goal is to show that there are $\Omega(\ell \cdot n^{2/3})$ such set-off/improving pairs. We first claim that the first and last edges of $P_{u,v}$ are missing in H_0. Seeking contradiction, assume that the first edge $\{u,u_1\} \in E(H_0)$, then the pair u_1,v has $W_{u_1,v} \le W_{u,v}$ and $d_G(u_1,v) < d_G(u,v)$ (using that the sub-path of $P_{u,v}$ from u_1 to v is the shortest path between u_1,v), and its stretch must be larger than $+(6+\varepsilon)W_{u,v}$ (otherwise u,v will have stretch at most $+(6+\varepsilon)W_{u,v}$ as well), so we should have considered the pair u_1,v before u,v, and added $P_{u_1,v}$ to H. That would produce a shortest path between u,v, which yields a contradiction to (2). A symmetric argument shows that the last edge is missing too.

Now, since H_0 contains a $n^{1/3}$-light initialization, but u (resp., v) has a missing edge, it follows that u (resp., v) has at least $n^{1/3}$ neighbors that are all lighter than the missing first (resp., last) edge of $P_{u,v}$, and thus of weight at most $W_{u,v}$. So there are at least $n^{1/3}$ choices for a and for c. By Lemma 1 there are at least $\Omega(\ell \cdot n^{1/3})$ choices for b. We conclude that there are at least $\Omega(\ell \cdot n^{1/3} \cdot n^{1/3}) = \Omega(\ell \cdot n^{2/3})$ pairs that are set-off/improved.

Let t be the number of edges added by all paths. Since every pair can be set-off only once, and improved $O(\frac{1}{\varepsilon})$ times, we get the following inequality

$$\Omega(t \cdot n^{2/3}) \le O(\frac{n^2}{\varepsilon}) \ ,$$

thus $t = O(\frac{n^{4/3}}{\varepsilon})$. ◀

4 A $+(2+\varepsilon)W$ subsetwise spanner

We will now show how to extend the technique of the $+(6+\varepsilon)W$ spanner to a $+(2+\varepsilon)W$ subsetwise spanner.

Let $G = (V,E,w)$ be a weighted undirected graph, a parameter $0 < \varepsilon < 1$, and $S \subseteq V$ a set of vertices. In this section we devise a $+(2+\varepsilon)W$ subsetwise spanner of size $O(n \cdot \sqrt{|S|}/\varepsilon)$. That is, the spanner guarantees an additive stretch at most $(2+\varepsilon) \cdot W_{u,v}$ for any $u,v \in S$.

Construction

Our algorithm follows a similar greedy idea to our previous constructions. We start by letting H be a $(\sqrt{|S|})$-light initialization of G. Next, sort all the pairs $\{u,v\} \in \binom{S}{2}$ by $W_{u,v}$ in increasing order, breaking ties arbitrarily. When considering u,v, we add $P_{u,v}$ to H if

$$d_H(u,v) > d_G(u,v) + (2+\varepsilon)W_{u,v}. \tag{4}$$

Analysis

Our main lemma is a variant of Lemma 4 tailored to the subsetwise case. For every path added to H, we improve the distance from many neighbors of the path to vertices in S, and have a good guarantee for all of them. Note that even though we claim improvements for many pairs in $S \times V$, the final spanner *does not* have guarantee for all such pairs, only to those in $S \times S$.

▶ **Lemma 6.** *Let $P_{u,v}$ be a path that was added to H. Denote by H_0 the spanner just before $P_{u,v}$ was added and by H_1 the spanner right after the path was added. Let a be a $(\sqrt{|S|})$-light neighbor of $x \in P_{u,v}$ with $w(a,x) \leq W_{u,v}$. Then both of the following hold.*

1. *$d_{H_1}(u,a) \leq d_G(u,a) + 2W_{u,v}$ **and** $d_{H_1}(v,a) \leq d_G(u,a) + 2W_{u,v}$.*
2. *$d_{H_1}(u,a) \leq d_{H_0}(u,a) - \frac{\varepsilon}{2}W_{u,v}$ **or** $d_{H_1}(v,a) \leq d_{H_0}(v,a) - \frac{\varepsilon}{2}W_{u,v}$.*

Proof. We begin with the first item. By the triangle inequality,

$$\begin{aligned} d_{H_1}(u,a) &\leq d_{H_1}(u,x) + d_{H_1}(x,a) \\ &= d_G(u,x) + d_G(x,a) \\ &\leq d_G(u,a) + d_G(x,a) + d_G(x,a) \\ &\leq d_G(u,a) + 2W_{u,v}. \end{aligned}$$

The bound on $d_{H_1}(v,a)$ follows in a symmetric manner, which concludes the proof of the first item.

Seeking contradiction, assume that the second item does not hold. This suggests that

$$d_{H_0}(u,a) < d_{H_1}(u,a) + \frac{\varepsilon}{2}W_{u,v} \leq d_G(u,x) + (1+\frac{\varepsilon}{2})W_{u,v} \ ,$$

and also

$$d_{H_0}(v,a) < d_{H_1}(v,a) + \frac{\varepsilon}{2}W_{u,v} \leq d_G(v,x) + (1+\frac{\varepsilon}{2})W_{u,v} \ .$$

So we have that

$$\begin{aligned} d_{H_0}(u,v) &\leq d_{H_0}(u,a) + d_{H_0}(a,v) \\ &< d_G(u,x) + (1+\frac{\varepsilon}{2})W_{u,v} + d_G(x,v) + (1+\frac{\varepsilon}{2})W_{u,v} \\ &= d_G(u,v) + (2+\varepsilon)W_{u,v}, \end{aligned}$$

which is a contradiction to (4), since we assumed that the path $P_{u,v}$ was added to the spanner. ◀

▶ **Theorem 7.** *For every undirected weighted graph $G = (V,E,w)$ with n vertices, a vertex set $S \subseteq V$ and a parameter $\varepsilon > 0$, there exists a deterministic polynomial time algorithm that produces a $+(2+\varepsilon)W$ subsetwise $S \times S$ spanner of size $O(\frac{1}{\varepsilon} \cdot n\sqrt{|S|})$.*

Proof. Our algorithm clearly yields a $+(2+\varepsilon) \cdot W$ spanner for $S \times S$, and can be done in polynomial time. It remains to bound the size of the spanner. The $(\sqrt{|S|})$-initialization adds at most $n \cdot \sqrt{|S|}$ edges to H.

Let $u,v \in S$ be such that $P_{u,v}$ is added to the spanner. Let H_0 be the spanner just before the path is added, and H_1 after. A pair (a,b) in $S \times V$ is said to *set-off* if this is the first time that $d_{H_1}(a,b) \leq d_G(a,b) + 2W_{u,v}$. This pair is *improved* if $d_{H_1}(a,b) \leq d_{H_0}(a,b) - \frac{\varepsilon}{2} \cdot W_{u,v}$.

By Lemma 1 if there are ℓ missing edges of $P_{u,v}$ in H_0, then there are at least $\Omega(\ell \cdot \sqrt{|S|})$ light neighbors that are connected to vertices on missing edges of $P_{u,v}$ with weight at most $W_{u,v}$. Thus there are $\Omega(\ell \cdot \sqrt{|S|})$ choices for a in Lemma 6. That is, so many pairs in $S \times V$ are set-off and improved. We notice that pairs from $S \times V$ can be set-off once and improved at most $\frac{4}{\varepsilon}$ times thereafter. If t is the total number of edges added to H by all the paths in the second stage of the algorithm, we get that

$$\Omega(t \cdot \sqrt{|S|}) \leq O(\frac{|S| \cdot |V|}{\varepsilon}) ,$$

thus $t = O(\frac{1}{\varepsilon} \cdot n\sqrt{|S|})$. ◀

5 A $+\tilde{O}(n^{\frac{1-\varepsilon}{2}}W)$ spanner of size $O(n^{1+\varepsilon})$

Let $G = (V, E, w)$ be a weighted undirected graph with n vertices, and let $0 \leq \varepsilon \leq 1$ be a parameter. We will now present our $+O(n^{\frac{1-\varepsilon}{2}} \log n)$ spanner of size $O(n^{1+\varepsilon})$.

Construction

Let H be a (n^{ε})-light initialization of G. We then add the edges of $(\log n, 0)$-greedy spanner from Lemma 2 to H. Next, we sort all the pairs $u, v \in V$ by $W_{u,v}$ in increasing order (breaking ties arbitrarily). For each pair (u, v) we add $P_{u,v}$ if

$$d_H(u, v) > d_G(u, v) + c \cdot n^{\frac{1-\varepsilon}{2}} \log n \cdot W_{u,v}, \tag{5}$$

where c is a constant to be determined.

Analysis

By the last step of the algorithm, every pair will have stretch $O(n^{\frac{1-\varepsilon}{2}} \log n \cdot W)$. The number of edges added by the (n^{ε})-light initialization of G is at most $n^{1+\varepsilon}$, and the $(\log n, 0)$-greedy spanner from Lemma 2 has $O(n)$ edges. The main difficulty of the analysis lies in bounding the number of edges in the paths added by the algorithm. Denote by $\mathcal{P}$ the set of paths added in the last stage. We start by bounding the number of such paths.

▶ **Lemma 8.** $|\mathcal{P}| \leq n^{\frac{1-\varepsilon}{2}}$.

Proof. We will define a labeling for the vertices. At the beginning, all the vertices will be unlabeled. Go over the added paths by the order of the algorithm. For every path $P_{x,y}$ which was added to the spanner, and every missing edge (a, b) in it, we label by $\{x, y\}$ all the unlabeled (n^{ε})-light neighbors of a and of b. We will show that for every added path, we label at least $n^{\frac{1+\varepsilon}{2}}$ vertices. This will imply that

$$|\mathcal{P}| \leq \frac{n}{n^{\frac{1+\varepsilon}{2}}} = n^{\frac{1-\varepsilon}{2}},$$

proving the lemma.

Seeking contradiction, assume that there is a path for which we labeled less than $n^{\frac{1+\varepsilon}{2}}$ vertices, and let $P_{u,v}$ be the first such path considered by the algorithm. Note that there can be at most $n^{\frac{1-\varepsilon}{2}}$ paths that were added before $P_{u,v}$.

Let H_0 be the spanner just before $P_{u,v}$ was added. The goal is to show a low stretch path in H_0 between u, v, contradicting the fact that $P_{u,v}$ was added. To this end, we distinguish between two types of edges in $P_{u,v}$ that are missing in H_0.

The first type are missing edges (a, b) that all the (n^{ε})-light neighbors of a or all the (n^{ε})-light neighbors of b are unlabeled. Observe that there is a constant k, so there can be at most $k \cdot n^{\frac{1-\varepsilon}{2}}$ such missing edges, since by Lemma 1 $k \cdot n^{\frac{1-\varepsilon}{2}}$ missing edges have at least $\Omega(k \cdot n^{\frac{1-\varepsilon}{2}} \cdot n^{\varepsilon}) = \Omega(k \cdot n^{\frac{1+\varepsilon}{2}})$ neighbors which are given labels. Choosing a large enough k, will contradict the assumption we label less than $n^{\frac{1+\varepsilon}{2}}$ vertices when adding $P_{u,v}$. Since there can't be many edges of this type, for each such edge (a, b) we can use the $\log n$-spanner which gives stretch at most $\log n \cdot w(a, b) \leq \log n \cdot W_{u,v}$. Thus the total stretch over all these edges is at most $k \log n \cdot n^{\frac{1-\varepsilon}{2}} \cdot W_{u,v}$.

The second type are missing edges with a labeled (n^{ε})-light neighbor. Suppose u' is a vertex in $P_{u,v}$ on a missing edge (u', u'') with an (n^{ε})-light neighbor labeled $\{x, y\}$. Let v' be the rightmost vertex on a missing edge (v'', v') in $P_{u,v}$ with an (n^{ε})-light neighbor labeled by $\{x, y\}$. Denote by a (resp. b) the light neighbor of u' (resp. v') with label $\{x, y\}$. Let x' (resp., y') be a vertex in $P_{x,y}$ such that a (resp., b) is a (n^{ε})-light neighbor of x' (resp., y') (see Figure 2). Note that $w(u', a) \leq w(u', u'') \leq W_{u,v}$, since the edge (u', u'') was not added in the (n^{ε})-initialization, and similarly $w(v', b) \leq W_{u,v}$. Also $w(x', a) \leq W_{x,y} \leq W_{u,v}$, since a got its label by being a light neighbor of a missing edge in P_{xy}, and $W_{x,y} \leq W_{u,v}$ by the initial sort of pairs according to the heaviest edge. Similarly $w(y', b) \leq W_{u,v}$. Recalling that all the edges to an (n^{ε})-light neighbor are in H_0, we can now see that the distance between u' and v' in H_0 has constant additive stretch:

$$\begin{aligned}
d_{H_0}(u', v') &\leq d_{H_0}(u', a) + d_{H_0}(a, x') + d_{H_0}(x', y') + d_{H_0}(y', b) + d_{H_0}(b, v') \\
&\leq d_G(u', a) + d_G(a, x') + d_G(x', y') + d_G(y', b) + d_G(b, v') \\
&\leq 2(d_G(u', a) + d_G(a, x')) + d_G(u', v') + 2(d_G(y', b) + d_G(b, v')) \\
&\leq d_G(u', v') + 8W_{u,v}.
\end{aligned}$$

We conclude that whenever we encounter a vertex u' on a missing edge with a light neighbor labeled $\{x, y\}$, we can simply use the path in H_0 to the last vertex v' on $P_{u,v}$ on a missing edge with a light neighbor labeled $\{x, y\}$, and pay only $8W_{u,v}$ additive stretch. Let z be the neighbor of v' closer to v, then use the multiplicative spanner in case the edge (v', z) is missing. The remaining path from z to v will clearly have no more missing edges with a light neighbor labeled $\{x, y\}$. Recall that we added at most $n^{\frac{1-\varepsilon}{2}}$ paths before $P_{u,v}$, so there can be at most $n^{\frac{1-\varepsilon}{2}}$ different labels. This suggests that the total additive stretch accumulated by the second type of missing edges is at most $(8 + \log n) \cdot n^{\frac{1-\varepsilon}{2}} \cdot W_{u,v}$.

Thus there exists a path in H_0 between u, v of length at most $d_G(u, v) + (8 + (1 + k) \log n) \cdot n^{\frac{1-\varepsilon}{2}} \cdot W_{u,v}$, setting $c \geq 9 + k$ contradicts the fact that $P_{u,v}$ was added by the algorithm. This concludes the proof of the lemma. ◀

▶ **Lemma 9.** *Adding $\mathcal{P}$ to H adds $O(n)$ edges to the spanner.*

Proof. Let $P_{u,v}$ be a path added by the algorithm. Let H_0 be the spanner just before it is added. Then for every edge $(a, b) \in P_{u,v}$ there are three cases:

1. At least one of the vertices a, b does not belong to any path previously added to H. Since every vertex has 2 edges touching it in the path, there can be at most $2n$ such edges among all the paths.
2. Both a, b belong to the same previously added path. Note that the edge (a, b) is already in H_0 in this case.
3. There is a previously added path $P_{x,y}$ such that $a \in P_{x,y}$ and $b \notin P_{x,y}$. Then the two paths $P_{x,y}$ and $P_{u,v}$ start their intersection at a.

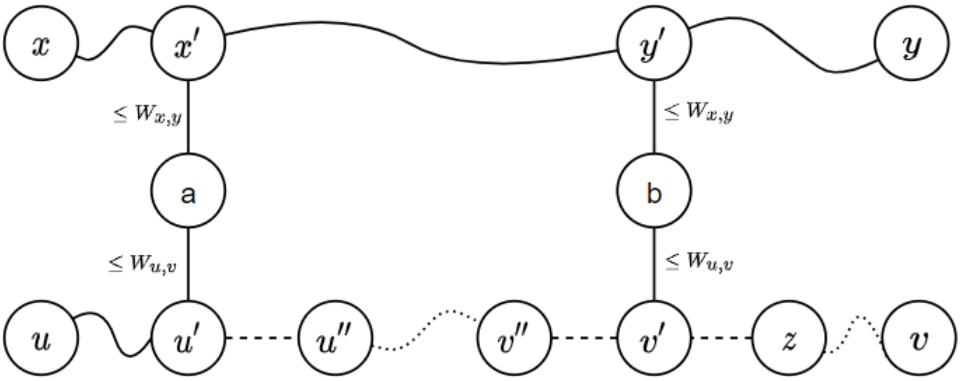


Figure 2 An illustration for Lemma 8. Straight lines and curved lines are edges and paths which are present in H_0. Dotted straight lines are edges missing in H_0 and dotted curved lines are path with possibly missing edges in H_0.

To bound the number of edges in case 3, note that every two paths can have only one intersecting subpath. So any pair of paths in $\mathcal{P}$ can introduce at most 2 edges to case 3 (the first and the last edge in their common subpath). By Lemma 8 there can be at most $2\binom{|\mathcal{P}|}{2} = O(n^{1-\varepsilon})$ such added edges in all the paths. ◀

By Lemma 9 the number of edges in H is $O(n^{1+\varepsilon})$. We have proven the following theorem.

▶ **Theorem 10.** *For every undirected weighted graph $G = (V, E, w)$ and $0 \leq \varepsilon \leq 1$, there exists a deterministic polynomial time algorithm that produces a $+O(n^{\frac{1-\varepsilon}{2}} \log n)W$ spanner of size $O(n^{1+\varepsilon})$.*

6 A +2W spanner in $\tilde{O}(n^2)$ time

In this section we present our the generalizing of +2 spanner construction algorithm of [12] for weighted graphs. Let $G = (V, E, w)$ be a weighted graph with n vertices, and fix $k = 1/2 \cdot \log n$ (assume k is an integer). Set $s_0 = n, s_1 = n/2, \ldots, s_k = n/2^k = \sqrt{n}$. For each $i = 0, 1, \ldots, k$, let V_i be the set of vertices of degree at least s_i (note that $V_0 = \emptyset$), set $V_{k+1} = V$. Let D_i be a set of vertices sampled independently at random from V, each with probability $p = \frac{c \log n}{s_i}$ for a constant $c > 1$. By standard considerations it follows that w.h.p. $|D_i| = \Theta(\frac{n \log n}{s_i})$, and D_i is a dominating set for V_i by Lemma 3.

For every $i \in [k]$, and for every $v \in V_i$, let $p_i(v) \in D_i$ be the closest vertex in D_i to v (breaking ties arbitrarily). Define $E_i^* = \{(v, p_i(v)) \ : \ v \in V_i\}$. Also, for every $v \in V_i$, define $Bunch_i(v) = \{(u, v) \in E \ : \ w((u, v)) < w((v, p_i(v)))\}$. For $v \notin V_i$, (i.e., $deg(v) < s_i$), set $Bunch_i(v) = \{(v, u) \in E\}$ to be the set of all edges incident on v.

Now set $E_1 = E$, and for each $i \in [2, k+1]$, set $E_i = \bigcup_{v \in V} Bunch_{i-1}(v)$. Note that for $v \in V_i$ the random variable $|Bunch_i(v)|$ is dominated by a geometric random variable with parameter p, so $\mathbb{E}[|Bunch_i(v)|] \leq \frac{s_i}{c \log n}$, thus for any $v \in V$, w.h.p. $|Bunch_i(v)| \leq O(s_i)$. We conclude that w.h.p. $|E_i| = O(n \cdot s_{i-1})$.

Construction

The algorithm is to add to the spanner H shortest path trees (SPT) from every vertex of D_i in the graph $(V, E_i \cup E_i^*)$, and take all edges of E_{k+1}. See Algorithm 1.

Algorithm 1 `+2W spanner`(G, S, ε).

```
Initialize $H \leftarrow \emptyset$;
for $i = 1, 2, \ldots, k$ do
    Build SPT trees rooted at every vertex $v \in D_i$ in $(V, E_i \cup E_i^*)$, and add them to $H$;
end for
return $H \cup E_{k+1}$;
```

We will also refer to each iteration i of this for-loop as *step i of the algorithm.*

Analysis of Size and Running Time

For every index $i \in [k]$, we have w.h.p. $|D_i| = \tilde{O}(n/s_i)$, thus $\sum_{i=1}^{k} |D_i| \cdot n = \tilde{O}(n^2) \cdot \sum_{i=1}^{k} \frac{1}{s_i} = \tilde{O}(n^{3/2})$. Also, w.h.p. $|E_{k+1}| \leq n \cdot s_k = \tilde{O}(n^{3/2})$. Hence the overall size of the spanner is $\tilde{O}(n^{3/2})$ as well.

To bound the running time, note that each step $i \in [k]$ of the algorithm requires computing $|D_i|$ SPTs in a graph with $O(|E_i| + n)$ edges. Using Dijkstra, each tree can be constructed in near linear time, so the total running time for step i is

$$\tilde{O}(|E_i| + n) \cdot |D_i| = \tilde{O}(n \cdot s_{i-1} \cdot n/s_i) = \tilde{O}(n^2)$$

time. The last step requires $\tilde{O}(|E|)$ time, and thus the overall time is $\tilde{O}(n^2)$.

Stretch Analysis

Let u, v be a vertex pair, let $P = P_{u,v}$ be the shortest $u - v$ path, and $W_{u,v}$ is the weight of the heaviest edge in P. For the sake of the following lemma, step 0 of the algorithm is before the algorithm starts.

▶ **Lemma 11.** *For every index $i = 0, 1, \ldots, k$, at least one of the following holds:*

1. $d_H(u, v) \leq d_G(u, v) + 2W_{u,v}$*, or*
2. $E(P) \subseteq E_{i+1}$.

Proof. The proof is by induction i.

Base ($i = 0$): Clearly $E(P) \subseteq E_1 = E$, i.e., the second assertion holds.

Step: Suppose that the induction hypothesis holds for some $i \in [0, k-1]$. If the first assertion holds for i, then obviously the first assertion holds for $i + 1$ as well. Hence, in this case we are done.

So suppose that the second assertion holds for i, i.e., $E(P) \subseteq E_{i+1}$. Consider the case that there exists an edge $e = (x, y) \in E(P) \setminus E_{i+2}$. (As otherwise $E(P) \subseteq E_{i+2}$, and the second assertion holds for $i + 1$.) Then we claim that both $x, y \in V_{i+1}$. To see this, assume that, e.g., $x \notin V_{i+1}$, but then by definition of Bunch for vertices not in V_{i+1} we have that $(x, y) \in Bunch_{i+1}(x) \subseteq E_{i+2}$, contradiction.

So we have $x, y \in V_{i+1}$, and $e = (x, y) \notin Bunch_{i+1}(y)$. Thus $y' = p_{i+1}(y)$ is defined, and

$$W_{u,v} \geq w((x, y)) \geq w((y, y')) = w((y, p_{i+1}(y))) \ .$$

Recall that $(y, p_{i+1}(y)) \in E^*_{i+1}$. So both paths $(y', y) \circ P(y, u)$ and $(y', y) \circ P(y, v)$ are contained in $E_{i+1} \cup E^*_{i+1}$. (We use $\circ$ here for concatenation, $P(y, u)$ for the subpath of P connecting y with u, and $P(y, v)$ for the subpath of P connecting y with v.)

Also, $y' \in D_{i+1}$. Hence inserting an SPT tree rooted at y' in $E_{i+1} \cup E^*_{i+1}$ into the spanner H guarantees

$$d_H(u, v) \leq d_G(u, v) + 2w(y', y) \leq d_G(u, v) + 2 \cdot W_{u,v} \ .$$

This tree is indeed inserted into the spanner on step $i + 1$, and so the first assertion for $i + 1$ holds. ◀

Apply the lemma for $i = k$. If the first assertion holds, then we are done. Otherwise $E(P) \subseteq E_{k+1}$. But then step $k + 1$ of the algorithm ensures that $d_H(u, v) = d_G(u, v)$, as all edges of E_{k+1} are inserted into H on this step. This completes the proof of the following theorem.

▶ **Theorem 12.** *Let $G = (V, E, w)$ be a weighted graph with n vertices, then there is an $\tilde{O}(n^2)$ time randomized algorithm that produces w.h.p. a $+2W$ spanner of size $\tilde{O}(n^{3/2})$.*

7 A $+4W$ emulator

In this section we present our the generalizing of $+4$ emulator of [12] for weighted graphs.

Construction

Our algorithm for a $+4W$ emulator works as follows. Start by letting $H = (V, E', d_G)$ be a $(2n^{1/3} \ln n)$-light initialization of G.[4] Let $S \subseteq V$ be a random set, created by sampling each vertex independently with probability $\frac{1}{n^{1/3}}$. We finish by adding $S \times S$ to E' (with weights corresponding to distances in G).

Analysis

▶ **Theorem 13.** *For every undirected weighted graph $G = (V, E, w)$, there exists a randomized algorithm that produces w.h.p. a $+4W$ emulator of size $O(n^{4/3} \log n)$.*

Proof. We begin with the stretch analysis. Let $u, v \in V$. If all the edges of $P_{u,v}$ exists in H, then $d_H(u, v) = d_G(u, v)$ and we are done.

Otherwise, let $u = x_1, x_2, \ldots x_k = v$ be the vertices of $P_{u,v}$ sorted by their distance from u. Let x_i, x_j be the first and last vertices for which $\{x_i, x_{i+1}\}, \{x_{j-1}, x_j\} \notin E'$.

We claim that each of x_i, x_j have at least $2n^{1/3} \ln n$ neighbors in G, because $\{x_i, x_{i+1}\}, \{x_{j-1}, x_j\}$ were not included in H as part of the light initialization. By Lemma 3, there exist $a, b \in S$ which are $(2n^{1/3} \ln n)$-light neighbors of x_i, x_j respectively. In addition, x_{i+1}, x_{j-1} are not $(2n^{1/3} \ln n)$-light neighbors of x_i, x_j, respectively, thus $w(x_i, a) \leq w(x_i, x_{i+1}) \leq W_{u,v}$ and $w(x_j, b) \leq w(x_{j-1}, x_j) \leq W_{u,v}$.

[4] By increasing the leading constant from 2 to c, we can reduce the failure probability to at most $O(n^{1-c})$.

The sub-paths $P_{u,x_i}, P_{x_j,v}$ exist in H, and also all the edges $\{x_i,a\},\{a,b\},\{b,x_j\} \in E'$. We can use them for bounding $d_H(u,v)$ (see figure 3).

$$\begin{aligned}
&d_H(u,v)\\
&\quad\le d_H(u,x_i)+d_H(x_i,a)+d_H(a,b)+d_H(b,x_j)+d_H(x_j,v)\\
&\quad= d_G(u,x_i)+d_G(x_i,a)+d_G(a,b)+d_G(b,x_j)+d_G(x_j,v)\\
&\quad\le d_G(u,x_i)+d_G(x_i,a)+d_G(x_i,a)+d_G(x_i,x_j)+d_G(b,x_j)+d_G(b,x_j)+d_G(x_j,v)\\
&\quad\le d_G(u,v)+4W_{u,v}.
\end{aligned}$$

Bounding the size is straightforward. The $n^{1/3}\log n$-light initialization introduces at most $O(n^{4/3}\log n)$ edges, while $|S|$ is a Bernoulli random variable with parameters $(n, \frac{1}{n^{1/3}})$. Therefore, $E[|S|] = n \cdot \frac{1}{n^{1/3}} = n^{2/3}$ and by Chernoff bound $|S| \le 2n^{2/3}$, w.h.p.. Thus $|S \times S| = O(n^{2/3} \cdot n^{2/3}) = O(n^{4/3})$ w.h.p..

Hence the total size of the emulator is $O(n^{4/3}\log n)$ w.h.p.. ◀

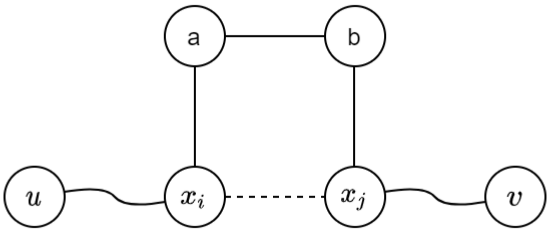


Figure 3 Straight lines are edges available in H. Curved lines are shortest paths available in H.

References

1 Amir Abboud and Greg Bodwin. The 4/3 additive spanner exponent is tight. *J. ACM*, 64(4):28:1–28:20, 2017. doi:10.1145/3088511.

2 Amir Abboud, Greg Bodwin, and Seth Pettie. A hierarchy of lower bounds for sublinear additive spanners. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 568–576, USA, 2017. Society for Industrial and Applied Mathematics.

3 Abu Reyan Ahmed, Greg Bodwin, Faryad Darabi Sahneh, Stephen G. Kobourov, and Richard Spence. Weighted additive spanners. In Isolde Adler and Haiko Müller, editors, *Graph-Theoretic Concepts in Computer Science - 46th International Workshop, WG 2020, Leeds, UK, June 24-26, 2020, Revised Selected Papers*, volume 12301 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2020. doi:10.1007/978-3-030-60440-0_32.

4 Donald Aingworth, Chandra Chekuri, Piotr Indyk, and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28(4):1167–1181, 1999. doi:10.1137/S0097539796303421.

5 Ingo Althöfer, Gautam Das, David P. Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discret. Comput. Geom.*, 9:81–100, 1993. doi:10.1007/BF02189308.

6 Surender Baswana, Telikepalli Kavitha, Kurt Mehlhorn, and Seth Pettie. New constructions of (alpha, beta)-spanners and purely additive spanners. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 672–681. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070526.

7 Greg Bodwin. Some general structure for extremal sparsification problems. *CoRR*, abs/2001.07741, 2020. arXiv:2001.07741.

8 Greg Bodwin and Virginia Vassilevska Williams. Better distance preservers and additive spanners. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 855–872. SIAM, 2016. doi:10.1137/1.9781611974331.ch61.

9 Gregory Bodwin and Virginia Vassilevska Williams. Very sparse additive spanners and emulators. In Tim Roughgarden, editor, *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 377–382. ACM, 2015. doi:10.1145/2688073.2688103.

10 Béla Bollobás, Don Coppersmith, and Michael Elkin. Sparse distance preservers and additive spanners. *SIAM J. Discret. Math.*, 19(4):1029–1055, 2005. doi:10.1137/S0895480103431046.

11 Shiri Chechik. New additive spanners. In Sanjeev Khanna, editor, *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 498–512. SIAM, 2013. doi:10.1137/1.9781611973105.36.

12 D. Dor, S. Halperin, and U. Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29:1740–1759, 2000.

13 M. Elkin. Computing almost shortest paths. In *Proc. 20th ACM Symp. on Principles of Distributed Computing*, pages 53–62, 2001.

14 M. Elkin and J. Zhang. Efficient algorithms for constructing $(1+\varepsilon,\beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18:375–385, 2006.

15 Michael Elkin, Yuval Gitlitz, and Ofer Neiman. Almost shortest paths with near-additive error in weighted graphs. *CoRR*, abs/1907.11422, 2019. arXiv:1907.11422.

16 Michael Elkin and Ofer Neiman. On efficient distributed construction of near optimal routing schemes. *Distributed Comput.*, 31(2):119–137, 2018. doi:10.1007/s00446-017-0304-4.

17 Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM J. Comput.*, 48(4):1436–1480, 2019. doi:10.1137/18M1166791.

18 Michael Elkin and David Peleg. (1+epsilon, beta)-spanner constructions for general graphs. *SIAM J. Comput.*, 33(3):608–631, 2004. doi:10.1137/S0097539701393384.

19 Mathias Bæk Tejs Knudsen. Additive spanners: A simple construction. In R. Ravi and Inge Li Gørtz, editors, *Algorithm Theory - SWAT 2014 - 14th Scandinavian Symposium and Workshops, Copenhagen, Denmark, July 2-4, 2014. Proceedings*, volume 8503 of *Lecture Notes in Computer Science*, pages 277–281. Springer, 2014. doi:10.1007/978-3-319-08404-6_24.

20 D. Peleg and A. Schäffer. Graph spanners. *J. Graph Theory*, 13:99–116, 1989.

21 D. Peleg and E. Upfal. A tradeoff between size and efficiency for routing tables. *J. of the ACM*, 36:510–530, 1989.

22 Seth Pettie. Low distortion spanners. *ACM Transactions on Algorithms*, 6(1), 2009. doi:10.1145/1644015.1644022.

23 M. Thorup and U. Zwick. Spanners and emulators with sublinear distance errors. In *Proc. of Symp. on Discr. Algorithms*, pages 802–809, 2006.

24 Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 1–10, New York, NY, USA, 2001. ACM. doi:10.1145/378580.378581.

Deterministic Size Discovery and Topology Recognition in Radio Networks with Short Labels

Adam Gańczorz
Institute of Computer Science, University of Wrocław, Poland

Tomasz Jurdziński
Institute of Computer Science, University of Wrocław, Poland

Mateusz Lewko
Institute of Computer Science, University of Wrocław, Poland

Andrzej Pelc
Département d'informatique, University of Québec en Outaouais, Gatineau, Canada

Abstract

We consider the fundamental problems of *size discovery* and *topology recognition* in radio networks modeled by simple undirected connected graphs. Size discovery calls for all nodes to output the number of nodes in the graph, called its size, and in the task of topology recognition each node has to learn the topology of the graph and its position in it.

We do not assume collision detection: in case of a collision, node v does not hear anything (except the background noise that it also hears when no neighbor transmits). The time of a deterministic algorithm for each of the above problems is the worst-case number of rounds it takes to solve it. Nodes have labels which are (not necessarily different) binary strings. Each node knows its own label and can use it when executing the algorithm. The length of a labeling scheme is the largest length of a label.

For size discovery, we construct a labeling scheme of length $O(\log\log\Delta)$ (which is known to be optimal, even if collision detection is available) and we design an algorithm for this problem using this scheme and working in time $O(\log^2 n)$, where n is the size of the graph. We also show that time complexity $O(\log^2 n)$ is optimal for the problem of size discovery, whenever the labeling scheme is of optimal length $O(\log\log\Delta)$. For topology recognition, we construct a labeling scheme of length $O(\log\Delta)$, and we design an algorithm for this problem using this scheme and working in time $O\left(D\Delta + \min(\Delta^2, n)\right)$, where D is the diameter of the graph. We also show that the length of our labeling scheme is asymptotically optimal.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases size discovery, topology recognition, radio network, labeling scheme

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.22

Related Version Due to the length requirements some proofs were omitted. For details see:
Full Version: `https://arxiv.org/abs/2105.10595` [13]

Funding *Adam Gańczorz*: Supported by the National Science Centre, Poland, grant 2017/25/B/ST6/02010.
Tomasz Jurdziński: Supported by the National Science Centre, Poland, grant 2017/25/B/ST6/02010.
Mateusz Lewko: Supported by the National Science Centre, Poland, grant 2017/25/B/ST6/02010.
Andrzej Pelc: Partially supported by NSERC discovery grant 2018-03899 and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 22; pp. 22:1–22:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Information about the topology of the network or some of its parameters, such as size, often determines the efficiency and sometimes the feasibility of many network algorithms. For example, graph exploration with stop performed in rings with non-unique labels is impossible without knowing some upper bound on the size of the ring. On the other hand, optimal broadcasting algorithms in wireless networks with distinct labels are faster when the topology of the network is known [21]. Hence, the problems of *size discovery* and *topology recognition* are fundamental in network computing. Size discovery calls for all nodes to output the number of nodes in the underlying graph, called its size, and in the task of topology recognition each node has to output an isomorphic copy of the graph with its position in it marked. More formally, in topology recognition, every node v of the graph G modeling the network must output a graph G' and a node v' in this graph, such that there exists an isomorphism $f : G \to G'$, for which $f(v) = v'$.

The model. We consider size discovery and topology recognition in radio networks modeled by simple undirected graphs. Throughout this paper $G = (V, E)$ denotes the graph modeling the network, n denotes the number of its nodes, D its diameter, and Δ its maximum degree. We use square brackets to indicate sets of consecutive integers: $[i, j] = \{i, \ldots, j\}$ and $[i] = [1, i]$.

As usually assumed in the algorithmic literature on radio networks, nodes communicate in synchronous rounds, starting in the same round. In each round a node can either transmit the same message to all its neighbors, or stay silent and listen. At the receiving end, a node v hears a message from a neighbor w in a given round, if v listens in this round, and if w is its only neighbor that transmits in this round. If more than one neighbor of a node v transmits in a given round, there is a *collision* at v. Two scenarios concerning collisions were considered in the literature. The availability of *collision detection* means that node v can distinguish collision from silence which occurs when no neighbor transmits. If collision detection is not available, node v does not hear anything in case of a collision (except the background noise that it also hears when no neighbor transmits). In this paper we do not assume collision detection. The time of a deterministic algorithm for each of the above problems is the worst-case number of rounds it takes to solve it.

If nodes are anonymous then neither size discovery nor topology recognition can be performed, as no communication in the network is possible. Indeed, without any labels, in every round either all nodes transmit or all remain silent, and so no message can be received. Hence we consider labeled networks. A *labeling scheme* for a given network represented by a graph $G = (V, E)$ is any function $\mathcal{L}$ from the set V of nodes into the set S of finite binary strings. The string $\mathcal{L}(v)$ is called the label of the node v. Note that labels assigned by a labeling scheme are not necessarily distinct. The *length* of a labeling scheme $\mathcal{L}$ is the maximum length of any label assigned by it. Every node knows a priori only its label, and can use it as a parameter for the size discovery or topology recognition algorithm.

Our goal is to construct short labeling schemes for size discovery and topology recognition in arbitrary radio networks, and to design efficient deterministic algorithms for each of these tasks, using such schemes. Such short schemes in the context of radio networks were studied for size discovery in [19], and for topology recognition in [18]. In [19] the authors worked in the model with collision detection. They constructed labeling schemes of length $O(\log \log \Delta)$ and a size discovery algorithm using this scheme and working in time $O(Dn^2 \log \Delta)$. They also proved that labels of size $\Omega(\log \log \Delta)$ are necessary to solve the size discovery problem in this model. In [18], the authors studied topology recognition without collision detection, similarly as we do in the present paper, but restricted attention only to tree networks. They constructed labeling schemes of length $O(\log \log \Delta)$ and a topology recognition algorithm working for arbitrary trees, using these schemes. Moreover, they showed that labels of size $\Omega(\log \log \Delta)$ are necessary to solve the topology recognition problem for trees.

Solving distributed network problems with short labels can be seen in the framework of algorithms with *advice*. In this paradigm that has recently got growing attention, an oracle knowing the network gives advice to nodes not knowing it, in the form of binary strings, and a distributed algorithm cooperating with the oracle uses this advice to solve the problem efficiently. The required size of advice (maximum length of the strings) can be considered as a measure of the difficulty of the problem. Two variations are studied in the literature: either the binary string given to nodes is the same for all of them [17] or different strings may be given to different nodes [8, 7, 11, 12], as in the case of the present paper. If strings may be different, they can be considered as labels assigned to nodes by a labeling scheme. Such labeling schemes permitting to solve a given network task efficiently are also called *informative labeling schemes*. One of the famous examples of using informative labeling schemes is to answer adjacency queries in graphs [2].

Several authors have studied the minimum amount of advice (i.e., label length) required to solve certain network problems (see the subsection Related work). The framework of advice or labeling schemes permits us to quantify the amount of information used to solve a network problem, such as size discovery or topology recognition, regardless of the type of information that is provided. It should be noticed that the scenario of the same advice (label) given to all nodes would be trivial in the case of radio networks: no communication could occur, and hence the advice would have to contain the size of the network for size discovery, and would not help for topology recognition, as nodes would not be able to find their position in the network without communicating.

Our results. It turns out that the optimal length of labeling schemes, both for size discovery and for topology recognition, depends on the maximum degree Δ of the graph. For size discovery, we construct a labeling scheme of length $O(\log \log \Delta)$, which is optimal, in view of [19], and we design an algorithm for this problem using this scheme and working in time $O(\log^2 n)$, where n is the size of the graph. We also show that time complexity $O(\log^2 n)$ is optimal for the problem of size discovery, whenever the labeling scheme is of optimal length $O(\log \log \Delta)$. Hence, without collision detection we achieve the same optimal length of the labeling scheme, as was done in [19] with collision detection, and for this optimal scheme our size discovery algorithm is exponentially faster than that in [19].

For topology recognition, we construct a labeling scheme of length $O(\log \Delta)$, and we design an algorithm for this problem using this scheme and working in time $O\left(D\Delta + \min(\Delta^2, n)\right)$, where D is the diameter of the graph. We also show that the length of our labeling scheme is asymptotically optimal, by proving that topology recognition in the class of arbitrary radio networks requires labeling schemes of length $\Omega(\log \Delta)$. (In fact we prove a stronger result that this lower bound holds even in the model with collision detection.) If the optimal length of a labeling scheme sufficient to solve a problem is considered a measure of the difficulty of the problem, our result shows, in view of the labeling scheme of length $O(\log \log \Delta)$ for topology recognition in trees [18], that this task is exponentially more difficult in arbitrary radio networks than in radio networks modeled by trees.

Related work. There is a vast literature concerning distributed algorithms for various tasks in radio networks. These tasks include, e.g., broadcasting [5, 15], gossiping [5, 14] and leader election [6, 22]. In some cases [5, 14], the authors use the model without collision detection, in others [16, 22], the collision detection capability is assumed.

Many authors use the framework of algorithms with advice (or equivalently informative labeling schemes) to investigate the amount of information needed to solve a given network problem. In [10], the authors compare the minimum size of advice required to solve two information dissemination problems, using a linear number of messages. In [11], it is shown that advice of constant size permits to carry out the distributed construction of a minimum spanning tree in logarithmic time. In [2], optimal labeling schemes are constructed in order to answer adjacency queries in graphs. In [9], the advice paradigm is used to solve online problems.

In the model of radio networks, apart from the previously mentioned papers [19] studying size discovery and [18] studying topology recognition in the framework of short labeling schemes, other authors studied the tasks of broadcast and multi-broadcast [20, 8, 7, 23], and convergecast [4] in this framework. While [8, 7] assume that short labels are given to anonymous nodes, [20] adopts a different approach. The authors study radio networks without collision detection for which it is possible to perform centralized broadcasting in constant time, i.e., when the topology of the network is known and all nodes have different labels. They investigate how many bits of additional information (i.e., not counting the labels of nodes) given to nodes are sufficient for performing broadcast in constant time in such networks, if the topology of the network is not known to the nodes.

The task of topology recognition was also investigated in models other than the radio model: in [12] the authors use the LOCAL model, and in [24] the model used is the congested clique.

2 Preliminaries

As our algorithms use recent results showing that there exist constant-length labeling schemes for the broadcast problem, we recall these results, adjust them to our needs and introduce an auxiliary notion called the *broadcast tree*. Then, using the existence of an efficient broadcast algorithm with constant-length labels, we give a lemma allowing to "encode" a given message M in labels of the neighborhood of some path. This path is associated with an efficient algorithm that collects the whole message in a single node and broadcasts it to the whole network.

We use the constant-length labeling schemes for the broadcast problem [8, 7]. In this problem, a source node s has a message which must be communicated to all other nodes. First, we recall the result regarding fast broadcast with constant-length labels from [7].

▶ **Theorem 1** ([7]). *There exists a labeling scheme of length $O(1)$ and an algorithm* Executor *using it which solves the broadcast problem in time $O(D \log n + \log^2 n)$.*

We say that a node v is *informed* in some round t of a broadcast algorithm if v knows the broadcast message in round t. Otherwise, v is *uninformed* in round t. We assume that feedback messages in the execution of Executor are distinct from the broadcast message (this can be easily ensured by adding a special sign to the broadcast message).

▶ **Definition 2** (Broadcast Tree). *Let $G = (V, E)$ be a graph with the source node $s \in V$.*

For each node $v \in V \setminus \{s\}$, the parent of v, denoted $parent(v)$, is the first node which successfully transmits the broadcast message to v during the execution of Executor *in G. The* broadcast tree *$T_{\text{Executor}(G)}$ is the tree with the root s and the set of edges $(v, parent(v))$ for each $v \neq s$.*

The level *of a node $v \in V$, denoted level(v), in the broadcast tree $T_{\textsc{Executor}(G)}$ is equal to the natural number i such that v receives the broadcast message from parent(v) for the first time in round i of the execution of* Executor *on G.*

In our algorithms for the size discovery problem, we will use the following properties of the algorithm Executor from [7].

▶ **Lemma 3.** *The broadcast algorithm* Executor *described in Theorem 1 satisfies the following properties:*

(1) *Assume that the level of a node v in the broadcast tree $T_{\textsc{Executor}(G)}$ is equal to i and the level of parent(v) in this tree is $j < i$. Then, parent(v) has a child with level k for each $k \in [j+1, i]$ such that $k \mod 3 = 1$.*

(2) *The maximum value of the levels of nodes of the broadcast tree $T_{\textsc{Executor}(G)}$ is larger than $t - 3$, where t is the number of rounds of the execution of* Executor*.*

(3) *Each node v can determine the level of parent(v) and its own level in the broadcast tree $T_{\textsc{Executor}}(G)$.*

We say that an algorithm A solves the *acknowledged broadcast* problem in T rounds iff it solves the broadcast problem, and moreover, all nodes of the graph know after T rounds that broadcast has been completed. Below, we show that Executor can be easily transformed into an acknowledged broadcast algorithm.

▶ **Corollary 4.** *Algorithm* Executor *can be transformed into algorithm* ExecAck *such that*

- ExecAck *solves the acknowledged broadcast problem on every graph G, using $O(1)$-bit labels and working in at most $3t$ rounds, where $t = O(D \log n + \log^2 n)$ is the number of rounds of the execution of* Executor *on G.*
- *after an execution of* ExecAck *on G, each node knows the number of levels of the broadcast tree $T_{\textsc{Executor}(G)}$, as well as its own level and the level of its parent in $T_{\textsc{Executor}(G)}$.*

Now, we make an observation regarding the length of labels sufficient to store (in a distributed way) an arbitrary message M initially unknown to all nodes, and subsequently make the message M known to all nodes.

▶ **Lemma 5.** *Let* ExecAck *be the acknowledged broadcast algorithm which satisfies Corollary 4. Then, an arbitrary message M of size m (initially unknown to nodes) can be made known to all nodes of graph G in $O(t)$ rounds using $O(1 + m/t)$-bit labels, where $t = O(D \log n + \log^2 n)$ is the number of rounds of the execution of* ExecAck *on G.*

3 Optimal Labeling Schemes

In this section we construct labeling schemes of optimal length for the size discovery and for the topology recognition problems, together with algorithms using these optimal schemes for these tasks. For the task of size discovery we will later show, in Section 5, a faster algorithm, in fact the fastest possible algorithm to solve the size discovery problem using a labeling scheme of optimal length.

3.1 Size discovery

In Sections 3.1.1–3.1.3 we design a general labeling scheme of length $O(\log \log \Delta)$ and a size discovery algorithm using this scheme, based on the broadcast algorithm working with constant-length labels [7].

3.1.1 $\log \Delta$-degree subtrees of a tree

We prove a general lemma stating that, given a tree T with maximum degree Δ and 3 bits of "local memory" at each node, and given a binary string M of length $\lceil \log(n+1) \rceil + 2$, there exists a subtree T' of T with maximum degree $\lfloor \log \Delta \rfloor + 1$, such that M can be split into the 3-bit local memories of the nodes of T'.

▶ **Lemma 6.** *Let $T = (V, E)$ be a tree with maximum degree Δ, size n and the root $r \in V$. Let M be a binary string of length $\lceil \log(n+1) \rceil + 1$. Then, there exists a subtree T' of T rooted at r and an assignment of binary strings to nodes of T' such that:*

(a) *the number of children of each node of T' is at most $\lfloor \log \Delta \rfloor + 1$,*

(b) *the string assigned to each node of T' has length at most 3,*

(c) *the string assigned to the root of T' has length 2,*

and the concatenation of all strings assigned to nodes of T' is M.

3.1.2 Labeling scheme

In this section we describe the labeling scheme CompactLabels which combines the so-called Δ-learning primitive from [19], the construction of a broadcast tree with help of the algorithm Executor from Theorem 1, and the limited-degree subtree described in Lemma 6.

The labels will consist of the root bit and three disjoint blocks: Δ-*block*, *BroadcastTree-block* and *Message-block*.

Δ-block and the root bit. First, we choose a node r with the largest degree Δ in the graph G, and mark r using the one-bit flag root. That is, $\text{root}_r = 1$ and $\text{root}_v = 0$ for each $v \neq r$.

Then, we describe $O(\log \log \Delta)$-bit strings called Δ-blocks which will be used to learn the value of Δ by the node r. The root r is assigned the pair $(x, 0)$, where x is the binary representation of the integer $\lfloor \log \Delta \rfloor + 1$. This integer is the size of the binary encoding of Δ. Then we choose $\lfloor \log \Delta \rfloor + 1$ neighbors of r and assign them consecutive natural numbers in the range $[1, \lfloor \log \Delta \rfloor + 1]$. The Δ-block of the ith chosen node is equal to the pair (a_i, b_i) where a_i is the binary representation of $i \in [\lfloor \log \Delta \rfloor + 1]$ of length $\lfloor \log(\lfloor \log \Delta \rfloor + 1) \rfloor + 1$, with leading zeros, and b_i is the ith bit of the binary representation of Δ. The Δ-blocks of other nodes are equal to the tuple $(0, 0)$. As the value of $\log \Delta$ can be encoded on $O(\log \log \Delta)$ bits, the length of Δ-blocks is $O(\log \log \Delta)$.

BroadcastTree-block. We apply Corollary 4 to construct the broadcast tree T_{Executor} of the graph with root (source vertex) r and assign $O(1)$-bit labels to nodes, used by the acknowledged broadcast algorithm ExecAck working in $O(D \log n + \log^2 n)$ rounds (Corollary 4).

Message-block. Let M be the binary representation of the size n of the graph. The message-block is the concatenation of two blocks: index and message.

We apply Lemma 6 in order to construct a subtree T' of T_{Executor} such that the degree of T' is not larger than $\lfloor \log \Delta \rfloor + 1$ and each node of T' is assigned a substring of M of length at most 3. To each node v of T' apart from the root we assign a natural number k_v such that the numbers assigned to the children of any node are consecutive integers starting from 0. For a node v of T', the block index of fixed length $O(\log \log \Delta)$ is the binary representation of the integer $k_v \in [1, \lfloor \log \Delta \rfloor] + 1$ (with leading zeros), where v is chosen as the child with number k_v of its parent. The substring of length at most 3 assigned to each node of T' according to Lemma 6 is the block message of this node. The concatenation of these substrings of

M forms the string M and substrings are assigned to nodes of T' in post-order, i.e., the substring assigned to a given node v is situated in M after the substrings assigned to the children of v in T', and substrings are assigned to the children w in increasing order of k_w. The blocks INDEX and MESSAGE of nodes outside of tree T' are the string (0).

Conceptually, the label of a node is the concatenation of the root bit, the Δ-block, the Broadcast Tree-block and the Message-block. In order to mark separations between the different blocks, we use the standard trick: the bit 1 is coded as 10, the bit 0 is coded as 01 and separations are coded as 00. This does not change the complexity of the label length, hence our labeling scheme has length $O(\log\log\Delta)$.

3.1.3 A simple size discovery algorithm

In this section we describe a simple size discovery algorithm using the above labeling scheme and working in time $O(D\log n + \log^2 n)$. We will show in Section 5 how to improve this algorithm to get the optimal time $O(\log^2 n)$.

We first design the size discovery algorithm AUXILIARYSD, which uses the labeling scheme described in the previous section. The algorithm AUXILIARYSD is a composition of three procedures, `Δ-learning`, `Ack-broadcast`, and `Size-learning`, corresponding to the three blocks of the labels described above: *Δ-block*, *BroadcastTree-block* and *Message-block*.
Procedure `Δ-learning`.
This procedure lasts $\lfloor\log\Delta\rfloor + 1$ rounds. In the ith round, the node with Δ-block (a_i, b_i) for $i > 0$ transmits the message with two bits: 0 and b_i. In each round all nodes except the root r (i.e., the node with the ROOT bit equal to 1) ignore messages with the value of the first bit equal to 0 and r stores consecutive bits b_i. The root r also knows the value of $\lfloor\log\Delta\rfloor + 1$ stored in its label, so it will know the number of rounds of the Δ-Learning procedure.
Procedure `Ack-broadcast`
We execute algorithm EXECACK from Corollary 4 with the source vertex r, the labels from the BroadcastTree-block and the broadcast message equal to the binary string encoding the value of Δ. As we use the algorithm solving the acknowledged broadcast problem, all nodes know the number of the last round of this procedure.
Procedure `Size-learning`
The goal of this procedure is to learn the binary representation M of the size n of the graph that is distributedly stored in nodes of the subtree T' of the broadcast tree T_{EXECUTOR}, as described in the previous section.

Let L be the number of levels of the broadcast tree T_{EXECUTOR}. The procedure `Size-learning` is split into phases $1, \ldots, L$ such that the strings MESSAGE stored in nodes from the level $L - i + 1$ are transmitted to their parents in phase i, together with messages containing strings MESSAGE received by those nodes from their subtrees. Each phase lasts $\lfloor\log\Delta\rfloor + 1$ rounds. At the beginning of the phase i, each node v of T' of level $L - i + 1$ reconstructs its message from all messages received in previous rounds and its own string MESSAGE. As the labeling scheme uses post-order encoding, the node v first concatenates the messages of its children in T' in the order of numbers assigned to them and then adds its own string MESSAGE at the end of the string to get M_v, the part of M stored in its subtree. Assume that the node v of level $L - i + 1$ has INDEX that is the binary representation of the integer $k_v \in [1, \lfloor\log\Delta\rfloor + 1]$. Then, v transmits M_v in the round k_v of phase i.

▶ **Lemma 7.** *The algorithm AUXILIARYSD solves the size discovery problem using a labeling scheme of length $O(\log\log\Delta)$, and it works in time $O(T_{\text{EXECUTOR}}(n, D) \cdot \log\Delta)$, where $T_{\text{EXECUTOR}}(n, D) = O(D\log n + \log^2 n)$ is the time of the broadcast algorithm EXECUTOR from Theorem 1.*

Now, we combine Lemma 7 with Lemma 5 to construct an improved version of AuxiliarySD, called GeneralSD. The goal is to get rid of the additional $\log \Delta$ multiplicative factor in the time complexity of a size discovery algorithm based on the broadcast algorithm Executor.

Given a graph G, we first compute the number t_G of rounds of the execution of Executor on G. If $t_G < \log n$, then we use the labeling scheme and the algorithm from Lemma 7. As the number of levels of the tree T_{Executor} is not larger than the number of rounds of the algorithm, the number of rounds of the size discovery algorithm is $O(t_G \log \Delta) = O(\log^2 n)$.

If $t_G \geq \log n$, we use the constant-length labeling scheme and the broadcasting algorithm ExecAck from Lemma 5, with the binary representation of the size n of the graph as the message M. Thus, we obtain an algorithm which solves the size discovery problem in time $O(D \log n + \log^2 n)$ using a labeling scheme of length $O\left(1 + \frac{\log n}{t_G}\right) = O(1)$.

Finally, in order to make the nodes of the graph aware of the chosen variant of the algorithm, we add one bit to all labels which contains the information whether or not $t_G < \log n$. Given the value of this bit, the nodes work according to instructions of the former or the latter algorithm described above. Thus, we obtain the following result.

▶ **Theorem 8.** *The algorithm GeneralSD solves the size discovery problem using a labeling scheme of length $O(\log \log \Delta)$, and works in time $O(D \log n + \log^2 n)$.*

3.2 Topology recognition

In this section we design a topology recognition algorithm working for any graph of maximum degree Δ using a labeling scheme of length $O(\log \Delta)$. We will later prove that this length is optimal. Our algorithm works in time $O\left(D\Delta + \min(n, \Delta^2)\right)$. First, in Section 3.2.1 we focus on constructing a BFS tree of the graph and efficient broadcast/gathering algorithms working in this tree. Then, in Section 3.2.2, the main topology recognition algorithm is presented, using the broadcast and gathering subroutines from Section 3.2.1.

3.2.1 Broadcast-gathering primitive

In this section we describe a labeling scheme of length $O(\log \Delta)$ and algorithms for the (acknowledged) *broadcast* and *gathering* problems using this scheme. Both algorithms work in time $O(D\Delta)$. We would like to emphasize that, although the broadcast problem can be solved more efficiently (cf. [7]) than the solution given here, our aim is to build the broadcast schedule associated with the gathering algorithm. Notice that the gathering problem requires a labeling scheme of lengths $\Omega(\log \Delta)$, e.g., if the communication graph's topology is a star.

Consider a graph $G = (V, E)$ of maximum degree Δ and diameter D, with a root node $r \in V$. In the gathering problem, each node $v \in V$ has some message M_v and the root r has to learn all messages M_v.

Denote by *layer*(v) the distance from r to a node v. We say that v is at layer i if *layer*$(v) = i$. Let V_i be the set of nodes at layer i. The neighborhood of a node v in G is denoted by $\mathcal{N}(v)$ and the set of neighbors of v at layer i is denoted by $\mathcal{N}_i(v)$. We fix an arbitrary strict total ordering on nodes denoted by $\prec$. For each node v, its label $\mathcal{L}(v)$ is equal to the tuple $(r_v, \textit{leaf}(v), a_v, b_v, g_v, \Delta)$, where r_v is the 1-bit flag indicating whether v is the root node r, *leaf*(v) is the bit indicating whether v has any neighbors on higher layers than *layer*(v), a_v is the bit used for acknowledged broadcast, while both b_v and g_v, called *broadcast-label* and *gather-label*, are special integer values used for broadcast and gathering respectively, described in more detail below. More precisely, the label $\mathcal{L}(v)$ is the concatenation of bits r_v, *leaf*(v), a_v, and binary representations of integers b_v, g_v and Δ.

The idea of the broadcast/gather labels and algorithms. For the broadcast problem, we split the nodes of each layer i into subsets $X_0, \ldots, X_{\Delta-1}$, where X_j is a maximal set of nodes of layer i not belonging to $\bigcup_{k<j} X_k$ such that the sets of neighbors of X_j at layer $i+1$ are pairwise disjoint, exluding the neighbors of nodes from $\bigcup_{k<j} X_k$. For $v \in X_j$ we set $b_v = j$. We observe that all nodes from X_j can transmit in the same round in order to deliver the broadcast message to all their neighbors at layer $i+1$, excluding the neighbors of the nodes from $\bigcup_{k<j} X_k$. This fact makes possible to deliver the broadcast message from layer i to layer $i+1$ in Δ rounds. For each node v at layer $i+1$, the first node which successfully transmits the broadcast message to v becomes $\mathit{parent}(v)$ and v becomes its child.

For the gathering problem, we assign distinct gather-labels in the range $[0, \Delta-1]$ to the children of each node. This assignment is required to satisfy also the following additional restriction. If u at layer $i+1$ is a neighbor of v at layer i and u is not a child of v, then the gather-label of u is not assigned to any child of v. With this restriction, the nodes at level i with a given value of the gather-label can transmit successfully messages to their parents in the same round. Thanks to this property, the values/messages stored in the nodes from the layer $i > 0$ can be gathered in their parents at layer $i-1$ in $O(\Delta)$ rounds.

More details, formal statement of the above mentioned properties and the final broadcast and gathering algorithms are described in the appendix, Section A.1.

3.2.2 Topology recognition algorithm

In order to solve the topology recognition problem, we choose an arbitrary node as the root r and assign to each node v the label which is the concatenation of two binary strings: the label $\mathcal{L}(v)$ used by our broadcast-gathering primitive and the binary representation of a natural number $\mathcal{C}(v)$ called the *color* of v which is the color of v in some fixed distance-two vertex coloring of the graph G with the set of colors equal to $[1, \Delta^2]$. Given these labels, we execute the following four-stage algorithm TopRec:

Stage 1. First, the acknowledged broadcast AckBrBFS is executed with the modification that each node v transmits the concatenation of the consecutive substrings g_u of the labels of nodes of the path from the root r to v, including g_v. This concatenation will be denoted by $\mathrm{ID}(v)$.

Stage 2. Then, in the block of Δ^2 rounds, each node v such that $\mathcal{C}(v) = i \in [1, \Delta^2]$ transmits in the round i of the block, sending the string $\mathrm{ID}(v)$, and each node stores received messages in its local memory.

▶ Remark. If $\Delta^2 > n$ then $\log \Delta \geq \frac{1}{2} \log n$. In this case, in order to reduce the number of rounds of Stage 2 to $\min(\Delta^2, n)$, we extend the labeling by assigning to each node a unique identifier in the range $[1, n]$ of length $O(\log n) = O(\log \Delta)$. Then, Stage 2 consists of n rounds, where the node with the identifier i transmits in round i of the stage.

Stage 3. Next, we execute the gathering algorithm GatherBFS, where the message M_v of a node v is equal to the set of IDs received by v in Stage 2, together with its own ID. The set of these messages, over all nodes v, permits each node to reconstruct the topology of the graph and situate itself in it.

Stage 4. Finally, we execute the broadcast algorithm BroadcastBFS, where the message M is equal to the set of messages of all nodes gathered at the root r in Stage 3.

▶ **Theorem 9.** *The algorithm* TopRec *solves the topology recognition problem on every graph, using a labeling scheme of length $O(\log \Delta)$. It works in time $O(D\Delta + \min(n, \Delta^2))$.*

4 Lower Bound on Lengths of Labels for Topology Recognition

In this section, we show that any algorithm for topology recognition in the class of general graphs requires a labeling scheme of length $\Omega(\log \Delta)$, and hence our algorithm TopRec from the previous section uses a labeling scheme of optimal length. The optimal length of a labeling scheme for topology recognition is thus exponentially larger than that sufficient to solve the easier problem of size discovery. In fact, we will prove a stronger result: the above lower bound holds even in the model with collision detection. Hence, till the end of this section we assume that collision detection is available.

The proof is divided into two parts. We first show the lower bound $\Omega(\log n)$ on the length of labeling schemes for topology recognition in graphs of degree $2^{\Omega(\log n)}$. Second, we generalize this result for graphs of arbitrary degree Δ, to get the lower bound $\Omega(\log \Delta)$.

In our proof, we exploit the fact that each node in the graph G has to learn its degree. Intuitively, if the set $\{\, deg(v) \mid v \in V \,\}$ of node degrees in G is larger than the set $\{\, \mathcal{L}(v) \mid v \in V \,\}$ of node labels then some two nodes $u, v \in V$, such that $deg(u) \neq deg(v) \wedge \mathcal{L}(u) = \mathcal{L}(v)$ have to learn their degrees through their *communication histories* $\mathcal{H}$. We will construct a family of graphs such that the labels have to be of length $\Omega(\log \Delta)$ so that nodes with the same labels and different degrees could have different communication histories.

We now formally define a *communication history* of a node for a given topology recognition algorithm executed on a given graph. Intuitively, this is a record of what the node learns during the execution of the algorithm (assuming collision detection).

▶ **Definition 10** (Communication history). *Let $G = (V, E)$ be a graph and let $\mathcal{L}$ denote a labeling scheme on G. Consider a topology recognition algorithm $\mathcal{A}$ executed on G and let i be a round of this algorithm. For each node $v \in V$, we define $\mathcal{H}_i(v)$ inductively such that $\mathcal{H}_i(v)$ denotes the communication history of v at the end of the ith round of $\mathcal{A}$.*

- *$\mathcal{H}_0(v) = \emptyset$, for all v.*
- *If*
  - *v transmits in the ith round, or*
  - *v listens and more than one neighbor of v transmits in the ith round (a collission),*

  we append the special character #, denoting "no message", to v's history, i.e., $\mathcal{H}_i(v) = [\mathcal{H}_{i-1}(v), \#]$,
- *If v listens and none of its neighbors transmits in the ith round (silence), we append ε to v's history, i.e., $\mathcal{H}_i(v) = [\mathcal{H}_{i-1}(v), \varepsilon]$.*
- *If v listens and successfully receives a message $m \in \{0,1\}^*$ in the ith round (exactly one neighbor of v transmits m), we append m to v's history, i.e., $\mathcal{H}_i(v) = [\mathcal{H}_{i-1}(v), m]$.*

$\mathcal{H}(v)$ denotes the communication history of v at the end of the last round of the execution of $\mathcal{A}$ on G.

4.1 The Lower Bound for Graphs with Large Degrees

In this section we define a family $\mathcal{G}$ of graphs of arbitrarily large size n with maximum degree $\Delta = 2^{\Omega(\log n)}$, such that a labeling scheme of length $\Omega(\log n)$ is necessary to solve the topology recognition problem on each suffciently large graph from this family.

Let n be a sufficiently large natural number such that $n^{1/2}$ is an even natural number. We construct an n-node graph $G_n = (V, E)$. The graph is composed of $n^{1/2}$ components, each component is composed of $n^{1/2}$ nodes. Let C_i denote the set of nodes from the ith component. Every node is connected with every node from a different component, i.e., for every C_i, C_j $(i \neq j)$ and every pair of nodes $u \in C_i, v \in C_j$ we have $(u, v) \in E$. Let $C(v)$ denote the component to which a node v belongs.

We now describe the set of edges connecting nodes of a component C_i composed of $k = n^{1/2}$ nodes. We divide nodes of C_i into two sets A and B, each of size $k/2$. Let $a_j \in A$ denote the jth node from A, and let $b_j \in B$ denote the jth node from B. Then, we connect a_j with the first j nodes from B: $(a_j, b_1), (a_j, b_2), \ldots, (a_j, b_j) \in E$. This concludes the construction of G_n. Observe that each G_n is connected. Note also that for any component, the set of different degrees of nodes in this component has size $k/2$. The degree of each node is in the range $[n - k + 1, n - k/2]$, since each component has size $n^{1/2}$. Note that nodes of different degrees that have equal labels and equal communication histories cannot learn that they have different degrees. The family $\mathcal{G}$ consists of all graphs G_n, such that $n^{1/2}$ is an even natural number.

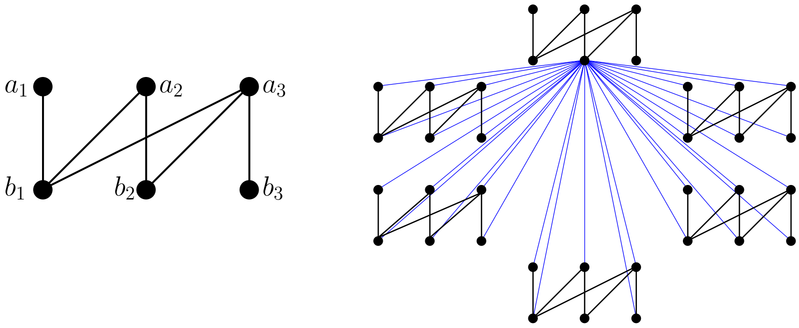


Figure 1 An illustration of the graph G_{36} from the family $\mathcal{G}$. The left picture depicts one component, the right picture depicts the whole graph. Blue edges connect nodes from different components. For clarity, only some of the inter-component edges are presented (each pair of nodes from different components should be connected by an edge).

Observe that, for each natural number n, there is at most one graph of size n in the family $\mathcal{G}$. Below, we make a simple but useful observation regarding lack of information received by most of the nodes in graphs from $\mathcal{G}$ if at least two nodes transmit in a round.

▶ **Fact 11.** *If at least two nodes $u \neq v$ transmit in the ith round then, for every node w such that $C(w) \neq C(u) \wedge C(w) \neq C(v)$, w does not hear any message in the ith round.*

We now define the *canonical history*, for a given graph G from the family $\mathcal{G}$ and a given topology recognition algorithm $\mathcal{A}$ executed on G. The canonical history is a specific communication history, which will be common to many nodes of G in the early stages of the execution of a topology recognition algorithm. It has the property that a node with the canonical history receives a message in a round i iff every node in the whole graph receives this message in this round.

▶ **Definition 12** (Canonical history). *Fix a graph G from the family $\mathcal{G}$, with a given labeling scheme $\mathcal{L}$, and a topology recognition algorithm $\mathcal{A}$ executed on G. With respect to G, $\mathcal{L}$ and $\mathcal{A}$, we define the canonical history $\widehat{\mathcal{H}}_i$ at the end of the ith round as follows:*

- $\widehat{\mathcal{H}}_0 = \emptyset$.
- $\widehat{\mathcal{H}}_i = [\widehat{\mathcal{H}}_{i-1}, m]$, *if there is exactly one node in G that transmits the message m in the ith round.*
- $\widehat{\mathcal{H}}_i = [\widehat{\mathcal{H}}_{i-1}, \#]$, *if at least two nodes of G transmit in round i,*
- $\widehat{\mathcal{H}}_i = [\widehat{\mathcal{H}}_{i-1}, \varepsilon]$, *if no node of G transmits in round i.*

We then define the set of components sharing the canonical history.

▶ **Definition 13.** *Fix a graph G from the family $\mathcal{G}$, with a given labeling scheme $\mathcal{L}$, and a topology recognition algorithm $\mathcal{A}$ executed on G. We say that a node v has the canonical history $\widehat{\mathcal{H}}_i$ if $\mathcal{H}_i(v) = \widehat{\mathcal{H}}_i$ at the end of the ith round of the execution of $\mathcal{A}$ on G. A component C_j has the canonical history at the end of the ith round if every node from that component has the canonical history at the end of this round.*

Let $\widehat{\mathcal{C}}_i$ denote the set of components of G that have the canonical history $\widehat{\mathcal{H}}_i$ at the end of the ith round:

$$\widehat{\mathcal{C}}_i = \{\, C_j \mid \forall_{v \in C_j} \mathcal{H}_i(v) = \widehat{\mathcal{H}}_i \,\}.$$

High-level idea of the proof of the lower bound. A node with the canonical history does not have any information that distinguishes it from other nodes with the canonical history, apart from its label. In topology recognition, nodes with different degrees must make different decisions (as each node has to situate itself in the graph), and there are at least $\frac{1}{2}\sqrt{n}$ different degrees in each component of a graph $G_n \in \mathcal{G}$. The communication history of all nodes at the beginning is equal to the canonical history. We show that, in every round, nodes from at most two components can change their histories from the canonical to a non-canonical one. Using this fact we show that some nodes change their histories from the canonical one to non-canonical after at least $\frac{1}{2}\sqrt{n}$ rounds. We also show that a node makes such a change (from the canonical to non-canonical history) in round i iff at least one node from its component (call it a *trigger*) transmits a message in that round and that nodes from at most one other component transmit messages in that round. Using these properties we show that, for each label l, at most two nodes with the label l can be triggers. On the other hand, we need a trigger in each component, so $\frac{1}{4}\sqrt{n}$ different labels are needed which implies that the length of the labeling scheme is $\Omega(\log n)$.

▶ **Corollary 14.** *Any algorithm solving the topology recognition problem in all graphs from the family $\mathcal{G}$ requires a labeling scheme of length $\Omega(\log n)$ on G_n.*

4.2 The Lower Bound for Graphs with Arbitrary Degrees

In this section we show that, for all positive integers $\Delta < n$, there exists a graph $H_{\Delta,n}$ of maximum degree $\Theta(\Delta)$ and of size $\Theta(n)$, such that any algorithm solving the topology recognition problem in $H_{\Delta,n}$ requires a labeling scheme of length $\Omega(\log \Delta)$. In order to prove this lower bound we use the family $\mathcal{G}$ of graphs G_n of maximum degrees $2^{\Omega(\log n)}$, where n is the size of the graph, constructed in Section 4.1.

Choose arbitrary positive integers $\Delta < n$. We construct a $\Theta(n)$-node graph $H_{\Delta,n}$ from $\Theta(n/\Delta)$ isomorphic copies of a graph G_k from $\mathcal{G}$, for some $k = \Theta(\Delta)$. Let k be the smallest natural number such that $k \geq \Delta$ and $\sqrt{k}$ is a natural even number. Let G'_Δ denote the graph obtained by the following slight modification of the graph G_k: add a new *special* node s connected to all other nodes of G_k. Let $G'^{(i)}_\Delta$, for all $i = 1, 2, \ldots, \lceil \frac{n}{\Delta} \rceil$, denote the ith isomorphic copy of the graph G'_Δ. All these copies are pairwise disjoint.

The graph $H_{\Delta,n}$ contains the above $\lceil \frac{n}{\Delta} \rceil$ disjoint isomorphic copies $G'^{(i)}_\Delta$ of G'_Δ, for all $i = 1, 2, \ldots, \lceil \frac{n}{\Delta} \rceil$. Moreover, we ensure connectivity of $H_{\Delta,n}$ in the following way. Let s_i denote the special node of $G'^{(i)}_\Delta$. We connect every s_i with s_{i+1} for each $i = 1, 2, \ldots, \lceil \frac{n}{\Delta} \rceil - 1$ by an edge, as well as $s_{\lceil \frac{n}{\Delta} \rceil}$ with s_1. Thus the special nodes of all graphs $G'^{(i)}_\Delta$ are connected in a ring. As the graph G'_Δ is connected, the graph $H_{\Delta,n}$ is connected as well. Moreover, $H_{\Delta,n}$ has maximum degree $\Theta(\Delta)$ and size $\Theta(n)$.

▶ **Theorem 15.** *For all sufficiently large positive integers $\Delta < n$, there exists a graph of size $\Theta(n)$ and of maximum degree $\Theta(\Delta)$ such that any topology recognition algorithm for this graph requires a labeling scheme with more than $\Delta^{1/4}$ distinct labels.*

The following corollary of Theorem 15 is the main result of this section.

▶ **Corollary 16.** *For all positive integers $\Delta < n$, there exists a graph of maximum degree $\Theta(\Delta)$ and of size $\Theta(n)$, such that any algorithm solving the topology recognition problem on this graph requires a labeling scheme of length $\Omega(\log \Delta)$.*

The above corollary shows that algorithm TopRec solves the topology recognition problem using a labeling scheme of optimal length.

5 Fast Algorithm for Size Discovery Problem

We finally design an algorithm for size discovery that is much faster than algorithm GeneralSD and also solves the size discovery problem using a labeling scheme of optimal length $O(\log\log \Delta)$. We also prove that this algorithm is the fastest possible among size discovery algorithms using such an optimal labeling scheme.

In Section 5.1, we describe a size discovery algorithm with time complexity $O(\log^2 n)$ using a labeling scheme of optimal length $O(\log\log \Delta)$. In Section 5.2, we show that this time complexity is asymptotically optimal for labeling schemes of length $O(\log\log \Delta)$.

5.1 The algorithm

In this section, we design the algorithm FastSD, which solves the size discovery problem in time $O(\log^2 n)$ using a labeling scheme with asymptotically optimal length $O(\log\log \Delta)$. Actually, if the diameter of the graph is $\Omega(\log n)$, constant-length labels are sufficient.

In Section 5.1.1, we generalize the recent labeling schemes for broadcast to the multi-source broadcast problem. Then, in Section 5.1.2, we give the idea of our new size discovery algorithm, using the multi-source broadcast algorithm. Finally, in Section A.2, we give details of our algorithm.

5.1.1 Multi-source broadcast

For the purpose of faster size discovery, we need a generalization of the result from [7] regarding the multi-source broadcast problem. Let $G = (V, E)$ be a graph and let a non-empty set $S \subset V$ be the set of *sources*. Assuming that all nodes from S know the broadcast message M, the goal of *multi-source broadcast* is to deliver the message M to all nodes of G. Let the *distance* from a node $v \in V$ to $S \subset V$ be the minimum of distances from v to s in G, over all $s \in S$. Moreover, let the diameter D_S of G with respect to S be the maximum of distances from v to S, over all $v \in V$.

We construct the multi-source broadcast algorithm MBroadcast through a simple modification of the broadcast algorithm Executor from [7]. The labeling scheme for the (single-source) broadcast algorithm from [7] is based on the probabilistic broadcast algorithm from [3]. As one can see in Section 6.3 of [7] (Lemmas 9 and 10), the probabilistic analysis depends merely on the length of a shortest path from a node which knows the broadcast message initially to a considered node v. More precisely, the only differences with respect to Executor concern the definition of the sets FRONTIER_1 and DOM_1, i.e., the set of uninformed nodes that have an informed neighbor before round 1 and the set of informed nodes that is a minimal dominating set of FRONTIER_1. As there is exactly one informed node s at the beginning in the broadcast problem, we have $\mathrm{DOM}_1 = \{s\}$ and $\mathrm{FRONTIER}_1$ is equal to the set of neighbors of s, in the labeling scheme used by Executor. For the multi-source broadcast problem, we

- set $\mathrm{FRONTIER}_1 = \{v \mid (s, v) \in E \text{ and } v \notin S\}$,
- choose a subset of S that is a minimum dominating set of $\mathrm{FRONTIER}_1$ and let this chosen subset of S be DOM_1.

Then, the analysis from [7] works also for the multi-source broadcast problem and it implies a multi-source broadcast algorithm MBroadcast using a constant-length labeling scheme and working in time $O(D_S \log n + \log^2 n)$.

▶ **Corollary 17.** *Algorithm MBroadcast solves the multi-source broadcast problem using a labeling scheme of length $O(1)$, and works in time $O(D_S \log n + \log^2 n)$, where S is the set of sources.*

Given the above multi-source broadcast algorithm, we can describe our faster size discovery algorithm. For this purpose we introduce a few auxiliary notions.

Let T_{BFS} be a BFS tree of the graph, where the root of the tree is an arbitrary node s. Moreover, let V_i be the set of nodes at distance i from the root, called the layer i of T_{BFS}. Thus, in particular, $V_0 = \{s\}$.

5.1.2 The idea of the $O(\log^2 n)$-time size discovery algorithm

If the diameter D of the graph G is $O(\log n)$, we simply apply GeneralSD which gives the desired $O(\log^2 n)$ time bound. Otherwise, we conceptually split the graph G into subgraphs of small diameter and then distribute information about the value of n in these subgraphs separately. Still, we have to assure that collisions caused by edges connecting different subgraphs do not prevent successful execution of our subroutines in the considered subgraphs. For brevity, we introduce the notation $\lg n = \lceil \log(n + 1) \rceil$, i.e., $\lg n$ is the length of the binary representation of the natural number n. We say that a layer V_i is *green* iff $\lfloor i/\lg n \rfloor$ is an even number; i.e., the layers $[j\lg n, (j + 1)\lg n - 1]$ are green for each even number $j \geq 0$. The nodes from layers $[j\lg n, (j + 1)\lg n - 1]$ for even j form the jth *stripe* X_j. Thus, we have the 0th, 2nd, 4th stripe and so on. Moreover, two consecutive stripes X_j and X_{j+2} are "separated" by nodes from the layers $V_{(j+1)\lg n}, V_{(j+1)\lg n+1}, \ldots, V_{(j+2)\lg n-1}$ which do not belong to any stripe. Nodes from each stripe are called *green*. Moreover, nodes from the last layer $V_{(j+1)\lg n-1}$ of the jth stripe for each even j are called *super-green*.

Our algorithm FastSD (an abbreviation of Fast Size Discovery) either executes GeneralSD (if $D = O(\log n)$) with appropriate labels or it consists of two stages:

Stage 1: In each stripe separately and in parallel, we execute an algorithm which ensures that, at the end of the stage, all super-green nodes from the stripe (i.e., nodes from the last layer $(j + 1)\lg n + 1$ of the stripe j) learn the value of n at the end of the stage. We perform this task in two phases. For the first phase, we choose a minimal set U_j of nodes from the first layer of the stripe (called a minimal BFS-cover) such that each super-green node in the stripe j is reachable from this set U_j, i.e., there is a BFS-path from a node of U_j to this node. Then, for each element of $u \in U_j$ we choose a path of length $\lg n$ such that it is a shortest path from u to the last layer $(j + 1)\lg n - 1$ of the stripe j. Moreover, we guarantee that those paths are conflict-free, i.e., there are no edges connecting nodes on different layers of different paths from the chosen family of paths. For each such path, the value of n is then encoded in 1-bit parts along the nodes of the path. In Phase 1 we "collect" these 1-bit parts in appropriate nodes from the BFS-cover U_j and, in Phase 2, we use the multi-source broadcast algorithm MBroadcast with U_j as the the set of sources and $D = \log n$ in order to broadcast the size n to super-green nodes (i.e., the nodes on the layer $(j+1)\lg n - 1$). As the consecutive stripes are "separated" by $\log n$ layers, we can execute Phase 1 separately in each stripe without the risk that interferences between nodes from different stripes cause any problem.

Stage 2: Execute the multi-source broadcast algorithm MBroadcast (see Corollary 17) with S equal to the set of all super-green nodes, i.e., S composed of the nodes from the layers $(j+1)\lg n - 1$, where j is even and $(j+1)\lg n - 1$ is not larger than the height of the BFS tree and $D_S = 2\lg n$. The broadcast message in the execution of MBroadcast is equal to the binary encoding of n, known to the nodes from S after Stage 1. As each node of the graph is at distance $\leq 2\lg n$ from some super-green node, all nodes learn the value of n in Stage 2.

5.2 Lower bound on time complexity of size discovery

We leverage a powerful technical result from [1], originally proved as a part of a proof of the $\Omega(\log^2 n)$ lower bound on time complexity of broadcast in radio networks, to show that time complexity of FastSD is asymptotically optimal.

Let $G = (U \cup V, E)$ be a bipartite graph, where U, V are the parts of the bipartition of G, i.e., $U \cap V = \emptyset$, and there are no edges between nodes inside U nor inside V, such that $|U| = |V| = n$ and there is no isolated vertex in $U \cup V$. A *bipartite broadcast schedule* for G is a sequence $U_1, \ldots, U_k$ of subsets of U such that, if one executes a k-round radio network algorithm in G with the set of transmitters in the ith round equal to U_i for each $i \in [k]$, then every node from V receives (at least) one message during this execution of the algorithm. The number of subsets k is called the *size* of the bipartite broadcast schedule for G.

▶ **Theorem 18** ([1]). *There exists a constant $c > 0$ such that, for each sufficiently large natural number n, there exists a bipartite graph G with sides of size n such that the size of each bipartite broadcast schedule for G is larger than $c \log^2 n$.*

Equipped with Theorem 18, we are ready to prove the lower bound $\Omega(\log^2 n)$ on the time complexity of the size discovery problem with short labels.

▶ **Theorem 19.** *The time complexity of any algorithm solving the size discovery problem on all graphs with size at most n, using labeling schemes of length smaller than $\frac{1}{4}\log n$, is $\Omega(\log^2 n)$.*

Since $O(\log\log\Delta)$ is $o(\log n)$ for any graph, and in view of the lower bound $\Omega(\log\log\Delta)$ from [19] on the length of labeling schemes permitting size discovery, Theorems 25 and 19 give the following corollary.

▶ **Corollary 20.** *Algorithm FastSD solves the size discovery problem on all graphs using a labeling scheme of asymptotically optimal length $O(\log\log\Delta)$, and it works in time $O(\log^2 n)$, which is aymptotically optimal for size discovery algorithms using asymptotically optimal labeling schemes.*

6 Conclusion

We constructed labeling schemes of asymptotically optimal length for size discovery and topology recognition in arbitrary radio networks without collision detection. We also designed algorithms to solve these problems using these optimal schemes.

In the case of size discovery, we showed that our algorithm using the labeling scheme of optimal length $O(\log\log\Delta)$ has also asymptotically optimal time among size discovery algorithms using optimal schemes for this problem. The main open problem left by our research is the following: What is the time of the fastest topology recognition algorithm using a labeling scheme of optimal length $O(\log\Delta)$?

References

1 Noga Alon, Amotz Bar-Noy, Nathan Linial, and David Peleg. A lower bound for radio broadcast. *J. Comput. Syst. Sci.*, 43(2):290–298, 1991.

2 Stephen Alstrup, Haim Kaplan, Mikkel Thorup, and Uri Zwick. Adjacency labeling schemes and induced-universal graphs. *SIAM J. Discret. Math.*, 33(1):116–137, 2019.

3 Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *J. Comput. Syst. Sci.*, 45(1):104–126, 1992.

4 Gewu Bu, Zvi Lotker, Maria Potop-Butucaru, and Mikael Rabie. Lower and upper bounds for deterministic convergecast with labeling schemes. Manuscript, HAL Id: hal-02650472, 2020.

5 Marek Chrobak, Leszek Gasieniec, and Wojciech Rytter. Fast broadcasting and gossiping in radio networks. *J. Algorithms*, 43(2):177–189, 2002. doi:10.1016/S0196-6774(02)00004-4.

6 Artur Czumaj and Peter Davies. Exploiting spontaneous transmissions for broadcasting and leader election in radio networks. *J. ACM*, 68(2):13:1–13:22, 2021. doi:10.1145/3446383.

7 Faith Ellen and Seth Gilbert. Constant-length labelling schemes for faster deterministic radio broadcast. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 213–222. ACM, 2020.

8 Faith Ellen, Barun Gorain, Avery Miller, and Andrzej Pelc. Constant-length labeling schemes for deterministic radio broadcast. In C. Scheideler and P. Berenbrink, editors, *31st Symposium on Parallelism in Algorithms and Architectures, SPAA 2019*, pages 171–178. ACM, 2019.

9 Yuval Emek, Pierre Fraigniaud, Amos Korman, and Adi Rosén. Online computation with advice. *Theor. Comput. Sci.*, 412(24):2642–2656, 2011. doi:10.1016/j.tcs.2010.08.007.

10 Pierre Fraigniaud, David Ilcinkas, and Andrzej Pelc. Communication algorithms with advice. *J. Comput. Syst. Sci.*, 76(3-4):222–232, 2010. doi:10.1016/j.jcss.2009.07.002.

11 Pierre Fraigniaud, Amos Korman, and Emmanuelle Lebhar. Local MST computation with short advice. *Theory Comput. Syst.*, 47(4):920–933, 2010. doi:10.1007/s00224-010-9280-9.

12 Emanuele G. Fusco, Andrzej Pelc, and Rossella Petreschi. Topology recognition with advice. *Inf. Comput.*, 247:254–265, 2016. doi:10.1016/j.ic.2016.01.005.

13 Adam Ganczorz, Tomasz Jurdzinski, Mateusz Lewko, and Andrzej Pelc. Deterministic size discovery and topology recognition in radio networks with short labels. *CoRR*, abs/2105.10595, 2021. arXiv:2105.10595.

14 Leszek Gasieniec, Aris Pagourtzis, Igor Potapov, and Tomasz Radzik. Deterministic communication in radio networks with large labels. *Algorithmica*, 47(1):97–117, 2007.

15 Leszek Gasieniec, David Peleg, and Qin Xin. Faster communication in known topology radio networks. *Distributed Comput.*, 19(4):289–300, 2007. doi:10.1007/s00446-006-0011-z.

16 Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. Randomized broadcast in radio networks with collision detection. In Panagiota Fatourou and Gadi Taubenfeld, editors, *PODC*, pages 325–334. ACM, 2013. doi:10.1145/2484239.2484248.

17 Christian Glacet, Avery Miller, and Andrzej Pelc. Time vs. information tradeoffs for leader election in anonymous trees. *ACM Trans. Algorithms*, 13(3):31:1–31:41, 2017.

18 Barun Gorain and Andrzej Pelc. Short labeling schemes for topology recognition in wireless tree networks. In S. Das and S. Tixeuil, editors, *24th International Colloquium, SIROCCO 2017*, volume 10641 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2017.

19 Barun Gorain and Andrzej Pelc. Finding the size of a radio network with short labels. In P. Bellavista and V. K. Garg, editors, *Proc. of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018*, pages 10:1–10:10. ACM, 2018.

20 David Ilcinkas, Dariusz R. Kowalski, and Andrzej Pelc. Fast radio broadcasting with advice. *Theor. Comput. Sci.*, 411(14-15):1544–1557, 2010. doi:10.1016/j.tcs.2010.01.004.

21 Dariusz R. Kowalski and Andrzej Pelc. Optimal deterministic broadcasting in known topology radio networks. *Distributed Computing*, 19(3):185–195, 2007. doi:10.1007/s00446-006-0007-8.

22 Dariusz R. Kowalski and Andrzej Pelc. Leader election in ad hoc radio networks: A keen ear helps. *J. Comput. Syst. Sci.*, 79(7):1164–1180, 2013. doi:10.1016/j.jcss.2013.04.003.

23 Colin Krisko and Avery Miller. Labeling schemes for deterministic radio multi-broadcast. *CoRR*, abs/2104.08644, 2021. arXiv:2104.08644.

24 Pedro Montealegre, Sebastian Perez-Salazar, Ivan Rapaport, and Ioan Todinca. Graph reconstruction in the congested clique. *J. Comput. Syst. Sci.*, 113:1–17, 2020.

A Appendix

A.1 Details of the Broadcast-gathering primitive

Assignment of broadcast-labels and gather-labels

Assignment of the values b_v and construction of a BFS-tree. For each layer i, we assign the values b_v as follows. First, we take any maximal subset X_0 of V_i such that the sets of neighbors at layer $i+1$ of elements of this set are pairwise disjoint. All nodes from $\mathcal{N}_{i+1}(v)$ will be called *children* of v, for each $v \in X_0$. Then, we construct sets $X_1, X_2, \ldots, X_{\Delta-1}$ as follows. Assume that the sets $X_0, \ldots, X_{j-1}$ are already constructed for $j > 0$. We define X_j as a maximal subset of nodes v at layer i such that

(a) $v \notin X_0 \cup \cdots \cup X_{j-1}$,

(b) for each pair of nodes $v \neq u$ such that $v, u \in X_j$, we have

$$\mathcal{N}_{i+1}(v) \cap \mathcal{N}_{i+1}(u) \setminus \left(\bigcup_{k=0}^{j-1} \bigcup_{x \in X_k} \mathcal{N}_{i+1}(x) \right) = \emptyset.$$

If a node $v \in X_j$ has a neighbor u in V_{i+1}, such that no node from X_k is in $\mathcal{N}_i(u)$ for all $k < j$, we say that v is the *parent* of u and u is a *child* of v. Observe that the set of edges determined by this parent-child relationship will form a BFS tree of the graph, denoted T_{BFS}. As we show in the full version of the paper, all nodes at layer i belong to $X_0 \cup \cdots \cup X_\Delta$. We set $b_v \leftarrow k$ for each node $v \in X_k$, where the sets $X_0, \ldots, X_\Delta$ are as above.

For the acknowledged broadcast, we choose some arbitrary leaf node with the highest value of *layer*(v) and a path P of length *layer*(v), such that P starts at r and ends at v in the tree T_{BFS} obtained through parent-child relationships described above. The value of a_u is set to 1 for each node u on the path P while the value a_w is set to 0 for each node w outside of P.

Assignment of the values g_v. As mentioned before, the bit g_v of each node v will be used in our gathering algorithm. The value g_r of the root node r is equal to 0. Then we assign values g_v layer by layer, assigning the values to nodes at layer i in phase i. In the ith phase, we order nodes at layer $i-1$ according to the ordering determined by the values of b_v, from the smallest to the largest and, among nodes with the same value of b_v, according to the ordering $\prec$. Let v_j be the jth node at layer $i-1$ in this order. We give the value g_u to each *child* u of v_j in the BFS-tree T_{BFS}, as follows. For a node v at layer $i-1$, we order its children at layer i (i.e., its neighbors at layer i which have not been assigned gather-labels yet) according to $\prec$, and assign to each of them the smallest non-negative integer not assigned earlier to any neighbor of v at layer i.

Below, we state some properties relevant for the gathering algorithm.

▶ **Lemma 21.** *The integers g_v have the following properties:*
(a) $g_v \in [0, \Delta - 1]$,
(b) $g_v \neq g_u$ *for each* $u \neq v$ *such that* $parent(u) = parent(v)$,
(c) *if* $parent(v) \neq parent(u)$ *and* $g_u = g_v$ *for some* $u \neq v$ *at the same layer, then* u *and* $parent(v)$ *are not connected by an edge.*

Since the label $\mathcal{L}(v)$ of any node is a concatenation of three bits and three representations of integers at most Δ, our labeling scheme has length $O(\log \Delta)$.

The properties of the labeling scheme facilitate the construction of a broadcast algorithm, acknowledged broadcast algorithm and a gathering algorithm, called BroadcastBFS, AckBrBFS and GatherBFS respectively (see the full version of the paper), with the following properties.

▶ **Lemma 22.** *The algorithms BroadcastBFS, AckBrBFS and GatherBFS solve the broadcast problem, the acknowledged broadcast problem and the gathering problem, respectively, using a labeling scheme of length $O(\log \Delta)$, and work in time $O(D\Delta)$.*

A.2 Details of algorithm FastSD

We say that a path $P = (x_1, \ldots, x_k)$ in a graph G with a fixed source node s is a *BFS-path* if, for each consecutive nodes x_i and x_{i+1} on P, the layer of x_{i+1} is by one larger than the layer of x_i. In other words, each edge of the path increases the distance from the source node s by one. Thus, in particular, we do not use edges connecting nodes from the same layer. We say that v is *BFS-reachable* from u if there exists a BFS-path from u to v.

In order to implement Stage 1 of FastSD, we first build a *BFS-cover* of stripe X_j by the nodes from the first layer $V_{j \log n}$ of the stripe, for each even j (see Figure 2).

▶ **Definition 23** (BFS-cover)**.** *For an even natural number j, the set of nodes $U_j \subseteq V_{j \log n}$ is a BFS-cover of the stripe $X_j = \bigcup_{i=jlg\ n}^{(j+1)lg\ n-1} V_i$ if, for each node $v \in V_{(j+1)lg\ n-1}$, there exists a BFS-path $u_0, u_1, \ldots, u_{lg\ n-1} = v$ such that $u_i \in V_{jlg\ n+i}$ for each $i \in [0, lg\ n-1]$, and $u_0 \in U_j$. In other words, the path $u_0, \ldots, u_{lg\ n-1}$ starts at some node from the given BFS-cover U_j and each edge of the path goes to the layer with larger index.*

We say that a BFS-cover U_j of the stripe X_j is a *minimal BFS-cover* of the stripe X_j if no proper subset U' of U_j is a BFS-cover of X_j. A set of BFS-paths $P_1, \ldots, P_k$ is *conflict-free* if, for each $i \neq j$, there is no edge (x_i, x_j) in the graph such that $x_i \in P_i$, $x_j \in P_j$, x_i and x_j belong to different layers.

▶ **Lemma 24.** *Let $U_j = \{u_1, \ldots, u_k\}$ be a minimal BFS-cover of the stripe X_j. Then, there exists a set of* conflict-free *BFS-paths $\{P_1, \ldots, P_k\}$ such that, for each $i \in [k]$:*
- *P_i starts at u_i,*
- *the final node of P_i is a super-green node of the stripe X_j,*

Let $M = M[1]M[2] \cdots M[\lg n]$ be the binary representation of n. In order to facilitate implementation of Stages 1 and 2 of FastSD, we construct labels of nodes in a given (green) stripe X in the following way (see Figure 2):

1. A minimal BFS-cover $U = \{u_1, \dots, u_k\}$ of the stripe X is chosen, along with the set of conflict-free paths $\mathcal{P} = \{P_1, \dots, P_k\}$ of length lg n, such that P_i starts at u_i and ends at a node in the last layer of X (i.e., at a supergreen node).
2. The set $X_{\text{BFS}} \subseteq X$ is determined such that $x \in X_{\text{BFS}}$ if x is BFS-reachable from some element of the BFS-cover U. (Observe that, although each node from the last layer of a stripe is BFS-reachable from U by the definition of a BFS-cover, it might be the case that nodes on smaller layers of the stripe are not reachable from U.)
3. The label of each node v is a concatenation of the following strings:
   a. a 4-bit string composed of flags REACH_v, SUPER-GREEN_v, COVER_v, PATHS_v indicating whether v is green and it belongs to X_{BFS} of its stripe X, whether v is super-green, whether v belongs to the chosen BFS-cover U of its stripe, and whether v belongs to one of the chosen cover-free paths $\mathcal{P} = \{P_1, \dots, P_k\}$ of its stripe, respectively;
   b. M_v: a one-bit string defined as follows. If v is the ith node of a path P_l from the above set of conflict-free paths, then M_v is equal to $M[i]$; otherwise, the value of M_v is 0;
   c. B_v: the constant-length label assigned to v by the labeling scheme for the multi-source broadcast algorithm MBROADCAST, with the graph $G' = (V', E')$, where $V' = X_{\text{BFS}}$, E' is the set of edges of G connecting nodes from X_{BFS}, and the set of sources S is equal to U.
   d. S2_v: the constant-length label assigned to v by the labeling scheme for the multi-source broadcast algorithm MBROADCAST, with the communication graph G, where the set of sources S is equal to the set of all super-green nodes (i.e., such nodes v that $\text{SUPER-GREEN}_v = 1$).

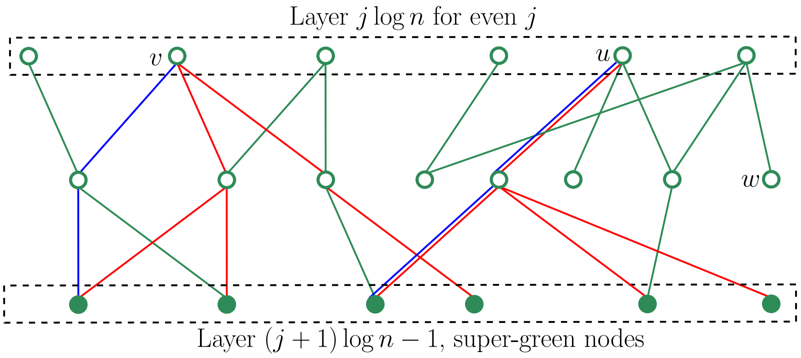

Layer $j \log n$ for even j
v
u
w
Layer $(j+1)\log n - 1$, super-green nodes

Figure 2 An example illustrating the notions and terms from the construction of FASTSD. The edges denote paths of length $\frac{1}{2}$lg n. The set $\{u, v\}$ is a minimal BFS-cover – the red edges show that each node from the layer $(j+1)$lg $n - 1$ is reachable from $\{u, v\}$ by a BFS-path. The blue edges form a set of conflict-free BFS-paths described in Lemma 24. The node w is an example of a node which is not active in Phase 2 of Stage 1, since $w \notin X_{\text{BFS}}$. However, w is at distance ≤ 2lg n from some super-green node from the layer $(j-1)$lg $n - 1$ (stripe $j - 2$) and therefore w receives a message with the size of the graph in Stage 2.

Using the above described labels, **Stage 1** for a stripe X is implemented as follows:

Phase 1: collecting the value of n at all nodes from the BFS-cover U.

- Round 1: each super-green node v from $\mathcal{P}$ (i.e., each v such that $\text{SUPER-GREEN}_v = 1$ and $\text{PATHS}_v = 1$) sends M_v.
- Rounds $2, 3, \dots,$ lg n: each node v with $\text{PATHS}_v = 1$ which received a message M in the previous round sends the concatenation of its bit M_v and the received message M.

Phase 2: broadcast of the value of n inside X_{BFS}.

Using the labels B_v, all nodes with $\text{REACH}_v = 1$ execute the multi-source broadcast algorithm MBroadcast, where the set of sources S is equal to U and the graph is $G' = (V', E')$, $V' = X_{\text{BFS}}$ (determined by the flags COVER_v) and E' is the set of edges of G connecting nodes from X_{BFS}.

Stage 2 is an execution of the multi-source broadcast algorithm MBroadcast on the whole graph G using the labels S2_v and thus with the source set S consisting of all super-green nodes (i.e., the nodes v with $\text{SUPER-GREEN}_v = 1$).

▶ **Theorem 25.** *Algorithm FastSD solves the size discovery problem in time $O(\log^2 n)$ using a labeling scheme of length $O(\log \log \Delta)$.*

Broadcast CONGEST Algorithms against Adversarial Edges

Yael Hitron
Weizmann Institute of Science, Rehovot, Israel

Merav Parter
Weizmann Institute of Science, Rehovot, Israel

Abstract

We consider the corner-stone broadcast task with an adaptive adversary that controls a fixed number of t edges in the input communication graph. In this model, the adversary sees the *entire* communication in the network and the random coins of the nodes, while maliciously manipulating the messages sent through a set of t edges (unknown to the nodes). Since the influential work of [Pease, Shostak and Lamport, JACM'80], broadcast algorithms against plentiful adversarial models have been studied in both theory and practice for over more than four decades. Despite this extensive research, there is no round efficient broadcast algorithm for *general* graphs in the CONGEST model of distributed computing. Even for a *single* adversarial edge (i.e., $t = 1$), the state-of-the-art round complexity is polynomial in the number of nodes.

We provide the first round-efficient broadcast algorithms against adaptive edge adversaries. Our two key results for n-node graphs of diameter D are as follows:

- For $t = 1$, there is a deterministic algorithm that solves the problem within $\widetilde{O}(D^2)$ rounds, provided that the graph is 3 edge-connected. This round complexity beats the natural barrier of $O(D^3)$ rounds, the existential lower bound on the maximal length of 3 edge-disjoint paths between a given pair of nodes in G. This algorithm can be extended to a $\widetilde{O}(tD^{O(t)})$-round algorithm against t adversarial edges in $(2t + 1)$ edge-connected graphs.
- For expander graphs with edge connectivity of $\Omega(t^2 \log n)$, there is a considerably improved broadcast algorithm with $O(t \log^2 n)$ rounds against t adversarial edges. This algorithm exploits the connectivity and conductance properties of G-subgraphs obtained by employing the Karger's edge sampling technique.

Our algorithms mark a new connection between the areas of fault-tolerant network design and reliable distributed communication.

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases CONGEST, Fault-Tolerant Network Design, Edge Connectivity

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.23

Related Version *Full Version*: https://arxiv.org/abs/2004.06436

Funding *Merav Parter*: This project is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949083).

Acknowledgements We are very grateful to David Peleg and Eylon Yogev for many useful discussions.

1 Introduction

Guaranteeing the uninterrupted operation of communication networks is a significant objective in network algorithms. The area of resilient distributed computation has been receiving a growing attention over the last years as computer networks grow in size and become more vulnerable to byzantine failures. Since the introduction of this setting by Pease et al. [50] and Lamport et al. [41, 50] distributed broadcast algorithms against various adversarial models have been studied in theory and practice for over more than four decades. Resilient distributed algorithms have been provided for broadcast and consensus [19, 20, 24, 12, 57,

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 23; pp. 23:1–23:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

55, 8, 56, 7, 23, 27, 25, 40, 52, 37, 21, 43, 34, 17, 38], as well as for the related fundamental problems of gossiping [9, 6, 13], and agreement [20, 50, 11, 16, 27]. See [51] for a survey on this topic. A key limitation of many of these algorithms is that they assume that the communication graph is the complete graph.

Our paper is concerned with communication graphs of arbitrary topologies. In particular, it addresses the following basic question, which is still fairly open, especially in the CONGEST model of distributed computing [54]:

▶ **Question 1.** *What is the cost (in terms of the number of rounds) for providing resilience against adversarial edges in distributed networks with arbitrary topologies?*

An important milestone in this regard was made by Dolev [19] who showed that the $(2t+1)$ node-connectivity of the graph is a necessary condition for guaranteeing the correctness of the computation in the presence of t adversarial nodes. Since then, byzantine broadcast algorithms for general graph topologies have been addressed mostly under simplified settings [52], e.g., probabilistic faulty models [53], cryptographic assumptions [26, 2, 1, 44], or under bandwidth-*free* settings (e.g., allowing neighbors to exchange exponentially large messages) [19, 43, 38, 40, 21, 38, 15]. A more in-depth comparison to the previous work can be found in the full paper [32]. In this paper, we consider the following extension of the standard CONGEST model to the adversarial setting:

The Adversarial CONGEST Model: The network is abstracted as an n-node graph $G = (V, E)$, with a processor on each node. Each node has a unique identifier of $O(\log n)$ bits. Initially, the processors only know the identifiers of their incident edges[a], as well as a polynomial estimate on the number of nodes n.

There is a *computationally unbounded* adversary that controls a fixed set of at most t edges in the graph, denoted hereafter as *adversarial* edges. The nodes do not know the identity of the adversarial edges, but they know the bound t. The adversary knows the graph topology and the random coins of the nodes. In each round, it is allowed to send $O(\log n)$ bit messages on each of the adversarial edges F (possibly a distinct message on each edge direction). It is *adaptive* as it can determine its behavior in round r based on the messages exchanged throughout the entire network up to round r.

[a] This is known as the standard KT1 model [5].

The definition naturally extends to adversarial *nodes* $F \subseteq V$ for which the adversary can send in each round, arbitrarily bad $O(\log n)$-bit messages on each of the edges incident to F. The primary complexity measure of this model is the *round* complexity. In contrast to many prior works in the adversarial setting, in our model, the nodes are *not* assumed to know the graph's topology, and not even its *diameter*. To address Question 1, we provide a comprehensive study of the adversarial broadcast problem, defined as follows:

The adversarial broadcast task: Given is a $(2t+1)$ edge-connected graph $G = (V, E)$ and a set $F \subset E$ of $|F| \leq t$ edges controlled by the adversary. There is a designated source node $s \in V$ that holds a message m_0. It is then required for all the nodes to output m_0, while ignoring all other messages.

To this date, all existing broadcast algorithms in the adversarial CONGEST model require a polynomial number of rounds, even when handling a single adversarial edge! Recently, Chlebus, Kowalski, and Olkowski [15] extended the result of Garay and Moses [27] to general

$(2t+1)$ node-connected graphs with minimum degree $3t$. Their algorithms, however, use *exponentially* large communication. Their message size can be improved to polynomial only when using authentication schemes (which we totally avoid in this paper). It is also noteworthy that the existing protocols for *node* failures might still require polynomially many rounds for general graphs, even for a single adversarial edge and for *small* diameter graphs.

A natural approach for broadcasting a message m_0 in the presence of t adversarial edges is to route the message along $(2t+1)$ edge-disjoint paths between the source node s, and each target node v. This allows each node to deduce m_0 by taking the majority message. This approach has been applied in previous broadcast algorithms (e.g., [19]) under the assumption that the nodes know *the entire graph*, and therefore can compute these edge disjoint paths. A recent work of [33] demonstrated that there are D-diameter $(2t+1)$ edge-connected graphs for which the maximal length of any collection of t edge-disjoint paths between a given pair of nodes might be as large as $(D/t)^{\Theta(t)}$. For $t = 3$, the length lower bound becomes $\Omega(D^3)$. Providing round efficient algorithms in the adversarial CONGEST model calls for a new approach.

Our approach in a nutshell. Our approach is based on combining the perspectives of fault-tolerant (FT) network design, and distributed graph algorithms. The combined power of these points of views allows us to characterize the round complexity of the adversarial broadcast task as a function of the graph diameter D, and the number of adversarial edges t. This is in contrast to prior algorithms that obtain a polynomial round complexity (in the number of nodes). On a high level, one of the main tools that we borrow from FT network design is the FT sampling technique [3, 60, 18, 30, 46, 49, 14], and its recent derandomization by [10, 36]. For a given graph G and a bound on the number of faults k, the FT sampling technique defines a small family $\mathcal{G} = \{G_i \subseteq G\}$ of G-subgraphs denoted as *covering family*, which is formally defined as follows:

▶ **Definition 2** ((L, k) covering family)**.** *For a given graph G, a family of G-subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ is an (L, k)* covering family *if for every $\langle u, v, F\rangle \in V \times V \times E^{\leq k}$ and any L-length u-v path $P \subseteq G \setminus F$, there exists a subgraph G_i such that (P1) $P \subseteq G_i$ and (P2) $F \cap G_i = \emptyset$.*

As the graph topology is unknown, one cannot hope to compute a family of subgraphs that are completely known to the nodes. Instead, we require the nodes to *locally* know the covering family in the following manner.

▶ **Definition 3** (Local Knowledge of a Subgraph Family)**.** *A family of ordered subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ where each $G_i \subseteq G$, is* locally known *if given the identifier of an edge $e = (u, v)$ and an index i, the nodes u, v can locally determine if $e \in G_i$.*

In the context of $(2t + 1)$ edge-connected graphs with t adversarial edges, we set $L = O(tD)$ and $k = O(t)$. The randomized FT sampling technique [60, 18] provides an (L, k) covering family $\mathcal{G}$ of cardinality $O(L^k \log n)$. [36] provided a deterministic construction with $O((L\,\mathrm{poly}(\log n))^{k+1})$ subgraphs.

One can show that by the properties of the covering family, exchanging the message m_0 over all subgraphs in $\mathcal{G}$ (in the adversarial CONGEST model) guarantees that all nodes successfully receive m_0. This holds since for every $v \in V$ and a fixed set of adversarial edges F, the family $\mathcal{G}$ contains a subgraph G_i which contains a short s-v path (of length L) and does not contain any of the adversarial edges. Our challenge is two folds:

1. provide a round-efficient algorithm for exchanging m_0 over all $\mathcal{G}$-subgraphs simultaneously,
2. guarantee that each node outputs the message m_0 while ignoring the remaining messages.

To address the first challenge, we show that the family of subgraphs obtained by this technique has an additional key property of *bounded width*. Informally, a family $\mathcal{G}$ of G-subgraphs has a bounded width if each G-edge appears in all but a bounded number of subgraphs in $\mathcal{G}$. The bounded width of $\mathcal{G}$ allows us to exchange messages in all these subgraphs simultaneously, in a nearly optimal number of rounds. The round complexity of this scheme is based on a very careful analysis which constitutes the key technical contribution in this paper. To the best of our knowledge, the bounded width property of the FT sampling technique has been used before only in the context of data structures [60, 31]. We find the fact that it finds applications in the context of reliable distributed communication to be quite exceptional. The second challenge is addressed by performing a second phase which filters out the corrupted messages. The round complexities of our broadcast algorithms for general graphs are dominated by the cardinality of covering families (which are nearly tight by [36]).

We also consider the family of expander graphs, which received a lot of attention in the context of distributed resilient computation [22, 58, 39, 4]. For these graphs, we are able to show covering families[1] of considerably smaller cardinality that scales linearly with the number of the adversarial edges. This covering family is obtained by using Karger's edge sampling technique, and its conductance-based analysis by Wulff-Nilsen [61]. We hope this result will also be useful in the context of FT network design. We next describe our key contribution in more detail.

1.1 Our Results

We adopt the gradual approach of fault tolerant graph algorithms, and start by studying broadcast algorithms against a single adversarial edge. Perhpas surprisingly, already this case has been fairly open. We show:

▶ **Theorem 4** (Broadcast against a Single Adversarial Edge). *Given a D–diameter, 3 edge-connected graph G, there exists a deterministic algorithm for broadcast against a single adversarial edge that runs in $\widetilde{O}(D^2)$ adversarial-*CONGEST *rounds. In addition, at the end of the algorithm, all nodes obtain a linear estimate for the* diameter *of the graph.*

This improves considerably upon the (implicit) state-of-the-art $n^{O(D)}$ bound obtained by previous algorithms (e.g., by [43, 15]). In addition, in contrast to many previous works (including [43, 15]), our algorithm does not assume global knowledge of the graph or any estimate on the graph's diameter. In fact, at the end of the broadcast algorithm, the nodes also obtain a linear estimate of the graph diameter.

Using the covering family obtained by the standard FT-sampling technique, it is fairly painless to provide a broadcast algorithm with a round complexity of $\widetilde{O}(D^3)$. Our main efforts are devoted to improving the complexity to $\widetilde{O}(D^2)$ rounds. Note that the round complexity of D^3 appears to be a natural barrier for this problem for the following reason. There exists a 3 edge-connected D-diameter graph $G = (V, E)$ and a pair of nodes s, v such that in any collection of 3 edge-disjoint s-v paths P_1, P_2, P_3, the length of the longest path is $\Omega(D^3)$ (By Corollary 40 of [33]). The improved bound of $\widetilde{O}(D^2)$ rounds is obtained by exploiting another useful property of the covering families of [36]. One can show that, in

[1] Using a somewhat more relaxed definition of these families

our context, each G-edge appears on all but $O(\log n)$ many subgraphs in the covering family. This plays a critical role in showing that the simultaneous message exchange on all these subgraphs can be done in $\widetilde{O}(D^2)$ rounds (i.e., linear in the number of subgraphs).

Multiple adversarial edges. We consider the generalization of our algorithms to support t adversarial edges. For $t = O(1)$, we provide broadcast algorithms with poly(D) rounds.

▶ **Theorem 5** (Broadcast against t-Adversarial Edges). *There exists a deterministic broadcast algorithm against t adversarial edges, for every D–diameter $(2t+1)$ edge-connected graph, with round complexity of $(tD\log n)^{O(t)}$. Moreover, this algorithm can be implemented in $O(tD\log n)$* LOCAL *rounds (which is nearly optimal).*

We note that we did not attempt to optimize for the constants in the exponent in our results for multiple adversarial edges. The round complexity of the algorithm is mainly dominated by the number of subgraphs in the covering family (extended to support t faults).

Improved broadcast algorithms for expander graphs. We then turn to consider the family of expander graphs, which has been shown to have various applications in the context of resilient distributed computation [22, 58, 39, 4]. Since the diameter of expander graphs is logarithmic, the algorithm of Theorem 5 yields a round complexity of $(t\log n)^{O(t)}$. In the full paper [32], we provide a considerably improved solution using a combination of tools. The improved broadcast algorithm is designed for $\Theta(t^2\log n/\phi)$ edge-connected ϕ-expander graphs.

▶ **Theorem 6.** *Given an n-node ϕ-expander graph with edge connectivity $\Omega(t^2\log n/\phi)$, there exists a randomized broadcast algorithm against t adversarial edges with round complexity of $O(t\cdot\log^2 n/\phi)$ rounds.*

To obtain this result, we use the edge sampling technique by Karger [35]. Karger showed that given a k edge-connected graph, when sampling each edge with a probability of $p = \Theta(1/t) = \Omega(\log n/\phi\cdot k)$, in the sampled subgraph all cuts are concentrated around their expectation. Wulff-Nilsen [61] showed that this sampled subgraph is an expander as well, w.h.p. We combine these properties to obtain a broadcast algorithm using $O(t\cdot\log^2 n)$ rounds. We believe that this technique might have other applications in the contexts of fault-tolerant network design and distributed secure computation (e.g., for the works of [48]).

Improved broadcast algorithms with a small shared seed. Finally, in the full paper [32], we show (nearly) optimal broadcast algorithms given that all nodes have a shared seed of $\widetilde{O}(1)$ bits.

▶ **Theorem 7** (Nearly Optimal Broadcast with Shared Randomness). *There exists a randomized broadcast algorithm against a single adversarial edge that runs in $\widetilde{O}(D)$ rounds, provided that all nodes are given* poly$(\log n)$ *bits of shared randomness.*

This result is obtained by presenting a derandomization for the well-known fault-tolerant (FT) sampling technique [60]. The FT-sampling technique is quite common in the area of fault-tolerant network design [18], and attracted even more attention recently [46, 14, 10, 36]. While it is relatively easy to show that one can implement the sampling using $\widetilde{O}(D)$ random bits, we show that $\widetilde{O}(1)$ bits are sufficient. This is obtained by using the pseudorandom generator (PRG) of Gopalan [29] and its recent incarnation in distributed settings [47]. We note that for a large number of faults t, the complexity is unlikely to improve from $D^{O(t)}$ to $\widetilde{O}(D)$ even when assuming shared randomness, i.e., the complexity can be improved only by a factor of D. Using the framework of *pseudorandom generator* [45, 59], we provide an improved broadcast algorithm for $\Omega(t \cdot \log n/\phi)$ edge-connected expander graphs that can tolerate t adversarial edges.

▶ **Lemma 8.** *Given an n-node ϕ-expander graph with edge connectivity $\Omega(t \cdot \log n/\phi)$, there exists a randomized broadcast algorithm against t adversarial edges, with a round complexity of $O(t \log^2 n/\phi)$, provided that all nodes have a shared seed of $O(\log n)$ bits.*

We hope that this work will motivate the study of additional distributed graph algorithms in the presence of adversarial edges and nodes.

Road Map. The broadcast algorithm with a single adversarial edge and the proof of Theorem 4 are given in Section 2. In Section 3 we consider multiple adversarial edges. Other results appear in the full paper [32].

Preliminaries. For a subgraph $G' \subseteq G$ and nodes $u, v \in V(G')$, let $\pi(u, v, G')$ be the unique u-v shortest path in G' where shortest-path ties are decided in a consistent manner. For a path $P = [u_1, \ldots, u_k]$ and an edge $e = (u_k, v)$, let $P \circ e$ denote the path obtained by concatenating e to P. Given a path $P = [u_1, \ldots, u_k]$ denote the sub-path from u_i to u_ℓ by $P[u_i, u_\ell]$. The asymptotic term $\widetilde{O}(\cdot)$ hides poly-logarithmic factors in the number of nodes n. Given a graph G, throughout we will use the following observation.

▶ **Observation 9.** *Consider an n-node D-diameter graph $G = (V, E)$ and let u, v be a pair of nodes that are connected in $G \setminus F$ for some $F \subseteq E$. It then holds that $\text{dist}_{G\setminus F}(u, v) \leq 2(|F| + 1) \cdot D + |F|$.*

Proof. Let T be a BFS tree in G rooted at some source s. The forest $T \setminus F$ contains at most $|F| + 1$ trees of diameter $2D$. Then, the u-v shortest path P in $G \setminus F$ can be transformed into a path P' containing at most $|F|$ edges of P as well as $|F| + 1$ tree subpaths of the forest $T \setminus F$. Therefore, $|P'| \leq 2(|F| + 1) \cdot D + |F|$ as desired. ◀

2 Broadcast Algorithms against an Adversarial Edge

In this section, we prove Theorem 4. We first assume, in Section 2.1 that the vertices have a linear estimate $c \cdot D$ on the diameter of the graph D, for some constant $c \geq 1$. A-priori, obtaining the diameter estimation seems to be just as hard as the broadcast task itself. In Section 2.2, we then show how this assumption can be removed. Throughout, we assume that the message m_0 consists of a single bit. In order to send a $O(\log n)$ bit message, the presented algorithm is repeated for each of these bits (increasing the round complexity by a $O(\log n)$ factor).

2.1 Broadcast with a Known Diameter

We first describe the adversarial broadcast algorithm assuming that the nodes have a linear estimate on the diameter D. In Section 2.2, we omit this assumption. The underlying objective of our broadcast algorithms is to exchange messages over reliable communication channels that avoid the adversarial edge e'. There are two types of challenges: making sure that all the nodes first *receive* the message m_0, and making sure that each node correctly *distinguish* between the true bit and the false one. Our algorithm BroadcastKnownDiam has two phases, a *flooding* phase and an *acceptance* phase, which at the high level, handles each of these challenges respectively.

The first phase propagates the messages over an ordered collection of G-subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ where each $G_i \subseteq G$ has several desired properties. Specifically, $\mathcal{G}$ is an (L, k) covering family (see Def. 2) for $L = O(D)$ and $k = 1$. An important parameter of $\mathcal{G}$ which determines the complexity of the algorithm is denoted as the *width*.

▶ **Definition 10** (Width of Covering Family). *The* width *of a collection of subgraphs $\mathcal{G} = \{G_1, \ldots, G_k\}$, denoted by $\omega(\mathcal{G})$, is the maximal number of subgraphs avoiding a fixed edge in G. That is,*

$$\omega(\mathcal{G}) = \max_{e \in G} |\{G_i \in \mathcal{G} \mid e \notin G_i\}| .$$

The broadcast algorithm starts by applying a 0-round procedure that provides each node in the graph with a local knowledge of an $(O(D), 1)$ covering family with bounded width. By [36], we have the following (see [32] for the proof):

▶ **Fact 11** ([36]). *Given a 3 edge-connected graph G, there exists a 0-round algorithm that allows all nodes to locally know an $(L, 1)$-covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ for $L = 7D$, such that $\ell = \widetilde{O}(D^2)$. The width of $\mathcal{G}$ is $\widetilde{O}(D)$.*

In the following we present a broadcast algorithm whose time complexity depends on several parameters of the covering family. This will establish the case where all nodes know a linear bound on the diameter D.

▶ **Theorem 12.** *Given a 3 edge-connected graph G of diameter D. Assuming that the nodes locally know an $(L, 1)$ covering family $\mathcal{G}$ for $L = 7D$, there exists a deterministic broadcast algorithm against an adversarial edge with $O(\omega(\mathcal{G}) \cdot L + |\mathcal{G}|)$ rounds.*

Broadcast with a known diameter (Proof of Theorem 12). Given a locally known $(L, 1)$ covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ for $L = 7D$, the broadcast algorithm has two phases. The first phase has $O(L \cdot \omega(\mathcal{G}) + |\mathcal{G}|)$ rounds and the second phase has $O(L)$ rounds.

Phase 1: Flooding phase. The flooding phase consists of $\ell = |\mathcal{G}|$ sub-algorithms $A_1, \ldots, A_\ell$, where in each algorithm A_i, the nodes propagate messages on the underlying subgraph $G_i \in \mathcal{G}$ that is defined locally by the nodes. The algorithm runs the subalgorithms $A_1, \ldots, A_\ell$ in a pipeline manner, where in the i'th round of sub-algorithm A_i, the source node s sends the message (m_0, i) to all its neighbors. For every i, a node $u \in V$, upon receiving a message (m', i) from a neighbor w for the first time, stores the message (m', i) and sends it to all its neighbors if the following conditions hold: (i) $(w, u) \in G_i$ and (ii) u did not receive a message (m', i) in a prior round[2]. For a node u and messages $(m_1, i_1), \ldots, (m_k, i_k)$ waiting to be sent in some round τ, u sends the messages according to the order of the iterations $i_1, \ldots, i_k$ (note that potentially $i_j = i_{j+1}$, and specifically, there might be at most two messages with index i_j, namely, $(0, i_j)$ and $(1, i_j)$).

[2] If it receives several (m', i) messages in the same round from different neighbors, it will be considered as only one.

Phase 2: Acceptance phase. The second phase consists of $O(L)$ rounds, in which *accept* messages are sent from the source s to all nodes in the graph as follows. In the first round, the source node s sends an *accept*(m_0) message to all its neighbors. Then every other node $u \in V$ accepts the message m' as its final output, and sends an *accept*(m') message to all neighbors, provided that the following conditions hold: (i) there exists $i \in \{1, \ldots, \ell\}$, such that u stored a message (m', i) in Phase 1; (ii) u received an *accept*(m') message in Phase 2 from a neighbor w_2, such that $(u, w_2) \notin G_i$. Since $\mathcal{G}$ is locally known, u can locally verify that $(u, w_2) \notin G_i$. This completes the description of the algorithm.

Correctness. We next prove the correctness of the algorithm. Missing proofs are deferred to the full paper [32]. We begin with showing that no node accepts a wrong message.

▷ Claim 13. No node $u \in V$ accepts a false message $m' \neq m_0$.

Proof. Assume towards contradiction there exists at least one node which accepts a message $m' \neq m_0$ during the second phase. Let u be the *first* node that accepts m'. By first we mean that any other node that accepted m', accepted the message in a later round than u, breaking ties arbitrarily. Hence, according to the algorithm, u received an *accept*(m') message from a neighbor w_1, and stored a message (m', i) in Phase 1, where $(u, w_1) \notin G_i$. Since u is the first node that accepts m', the node w_1 did not accept m' in the previous round. We conclude that the edge (w_1, u) is the adversarial edge and all other edges are reliable. Because the adversarial edge (w_1, u) was not included in the i'th graph G_i, all messages of the form (m', i) sent by the adversarial edge in Phase 1 are ignored. Since all other edges are reliable, no node received (and did not ignore) the false message (m', i) during the first phase - in contradiction to the assumption that u stored a message (m', i) in Phase 1, and therefore received the message (m', i) in Phase 1. ◁

From Claim 13 we can conclude that in the case where the adversarial edge initiates a false broadcast, it will not be accepted by any of the nodes.

▶ **Corollary 14.** *In case $e' = (v_1, v_2)$ initiates the broadcast, no node accepts any message.*

Proof. Since no node initiated the broadcast, in the second phase the only nodes that can receive *accept*(m) messages are v_1 and v_2 over the edge e'. In addition, since e' also initiates the first phase, for every node storing a message (m, i) in Phase 1 it must hold that $e' \in G_i$. Hence, we can conclude that neither v_1 nor v_2 accepts any of the false messages. Consequently, no node in $V \setminus \{v_1, v_2\}$ receives an *accepts*(m) message for any m, as required. ◀

So far, we showed that if a node v accepts a message, it must be the correct one. It remains to show that each node indeed accepts a message during the second phase. Towards that goal, we will show the collection of ℓ sub-algorithms executed in Phase 1 can be simulated in $O(\omega(\mathcal{G}) \cdot L + |\mathcal{G}|)$ rounds. This argument holds regardless of the power of the adversary.

▶ **Lemma 15.** *Consider an $(L, 1)$ covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ for G that is locally known by all the nodes. For a fixed node v, an edge e, and an L-length s-v path $P \subseteq G \setminus \{e, e'\}$, let $G_i \in \mathcal{G}$ be the subgraph containing P where $e \notin G_i$. Then, v receives the message (m_0, i) in Phase 1 within $O(L \cdot \omega(\mathcal{G}) + |\mathcal{G}|)$ rounds of that phase.*

We note that by Observation 9 taking $L = 7D$ yields that for every node v and edge e, it holds that $\text{dist}_{G \setminus \{e, e'\}}(s, v) \leq L$. Hence, by the properties of the covering family $\mathcal{G}$, for every node v and an edge e there exists an L-length s-v path $P \subseteq G \setminus \{e, e'\}$ and a subgraph G_i that contains P and avoids e. The proof of Lemma 15 is one of the most technical parts in this paper. Whereas pipeline is a very common technique, especially in the context of broadcast algorithms, our implementation of it is quite nontrivial. Unfortunately, since our adversary has a full knowledge of the randomness of the nodes, it is unclear how to apply the random delay approach of [28, 42] in our setting. We next show that our pipeline approach works well thanks to the bounded width of the covering family.

Proof of Lemma 15. Let $P = (s = v_0, \dots, v_\eta = v)$ be an s-v path in G_i where $\eta \leq L$. For simplicity, we consider the case where the only message propagated during the phase is m_0. The general case introduces a factor of 2 in the round complexity. This holds since there could be at most two messages of the form $(0, i)$ and $(1, i)$. We also assume, without loss of generality, that each node v_j receives the message (m_0, i) for the *first* time from v_{j-1}. If v_j received (m_0, i) for the first time from a different neighbor in an earlier round, the time it sends the message can only decrease.

In order to show that $v_\eta = v$ receives the message (m_0, i) within $O(L \cdot \omega(\mathcal{G}) + |\mathcal{G}|)$ rounds, it is enough to bound the total number of rounds the message (m_0, i) spent in the *queues* of the nodes of P, waiting to be sent. That is, for every node $v_j \in P$ let r_j be the round in which v_j received the message (m_0, i) for the first time, and let s_j be the round in which v_j sent the message (m_0, i). In order to prove Lemma 15, our goal is to bound the quantity $T = \sum_{j=1}^{\eta-1}(s_j - r_j)$.

For every $k < i$ we denote the set of edges from P that are not included in the subgraph G_k by $N_k = \{(v_{j-1}, v_j) \in P \mid (v_{j-1}, v_j) \notin G_k\}$, and define $\mathcal{N} = \{(k, e) \mid e \in N_k, k \in \{1, \dots, i-1\}\}$. By the definition of family $\mathcal{G}$, it holds that:

$$|\mathcal{N}| = \sum_{k=1}^{i-1} |N_k| \leq \eta \cdot \omega(\mathcal{G}) = O(\omega(\mathcal{G}) \cdot L).$$

For every node $v_j \in P$ let Q_j be the set of messages (m_0, k) that v_j sent between rounds r_j and round s_j. By definition, $|Q_j| = s_j - r_j$, and $T = \sum_{j=1}^{\eta-1} |Q_j|$.

Thus, to prove Lemma 15 its enough to show that $\sum_{j=1}^{\eta-1} |Q_j| \leq |\mathcal{N}|$. This is shown next in two steps. First we define a set I_j consisting of certain (m_0, k) messages such that $|Q_j| \leq |I_j|$, for every $j \in \{1, \dots, \eta - 1\}$. Then, we show that $\sum_{j=1}^{\eta} |I_j| \leq |\mathcal{N}|$.

Step one. For every node $v_j \in P$, let I_j to be the set of messages (m_0, k) satisfying the following three properties : (1) $k < i$, (2) v_j sent the message (m_0, k) before sending the message (m_0, i) in round s_j, and (3) v_j did not receive the message (m_0, k) from v_{j-1} before receiving the message (m_0, i) in round r_j. In other words, the set I_j includes messages received by v_j, with a graph index at most $i - 1$, that are either received from v_{j-1} between round r_j and round s_j, or received by v_j from another neighbor $w \neq v_{j-1}$ by round s_j (provided that those messages were not received additionally from v_{j-1}). Note that it is not necessarily the case that $Q_j \subseteq I_j$, but for our purposes, it is sufficient to show the following.

▷ Claim 16. For every $1 \leq j \leq \eta - 1$ it holds that $|Q_j| \leq |I_j|$.

Step two. We next show that $\sum_{j=1}^{\eta-1} |I_j| \leq |\mathcal{N}|$ by introducing an injection function f from $\mathcal{I} = \{(v_j, k) \mid (m_0, k) \in I_j\}$ to $\mathcal{N}$, defined as follows. For $(v_j, k) \in \mathcal{I}$, set $f((v_j, k)) = (k, (v_{h-1}, v_h))$ such that (v_{h-1}, v_h) is the closest edge to v_j on $P[v_0, v_j]$ where $(v_{h-1}, v_h) \in N_k$ (i.e., $(v_{h-1}, v_h) \notin G_k$).

$$f((v_j, k)) = (k, (v_{h-1}, v_h)) \mid h = \min_{\tau < i}\{t \mid (v_{\tau-1}, v_\tau) \in N_k\} .$$

We begin by showing the function is well defined.

▷ Claim 17. The function $f : \mathcal{I} \to \mathcal{N}$ is well defined.

Next, we show the function f is an injection.

▷ Claim 18. The function f is an injection.

Proof. First note that by the definition of the function f, for every $k_1 \neq k_2$, and $1 \leq j_1, j_2 \leq \eta - 1$ such that $(v_{j_1}, k_1), (v_{j_2}, k_2) \in \mathcal{I}$ it holds that $f((v_{j_1}, k_1)) \neq f((v_{j_2}, k_2))$. Next, we show that for every $k < i$ and $1 \leq j_1 < j_2 \leq \eta - 1$ such that $(v_{j_1}, k), (v_{j_2}, k) \in \mathcal{I}$, it holds that $f((v_{j_1}, k)) \neq f((v_{j_2}, k))$. Denote $f((v_{j_2}, k)) = (k, (v_{h_2-1}, v_{h_2}))$ and $f((v_{j_1}, k)) = (k, (v_{h_1-1}, v_{h_1}))$. We will now show that $(v_{h_2-1}, v_{h_2}) \in P[v_{j_1}, v_{j_2}]$. Since $(v_{h_1-1}, v_{h_1}) \in P[v_0, v_{j_1}]$, it will then follow that $(v_{h_1-1}, v_{h_1}) \neq (v_{h_2-1}, v_{h_2})$.

Assume towards contradiction that $(v_{h_2-1}, v_{h_2}) \in P[v_0, v_{j_1}]$. By the definition of f, we have $P[v_{i_1}, v_{i_2}] \subseteq G_k$. Since $(v_{j_1}, k) \in \mathcal{I}$, the node v_{j_1} sent the message (m_0, k) before sending (m_0, i). Additionally, since by our assumption, every node $v_t \in P$ receives the message (m_0, i) for the first time from node v_{t-1} (its incoming neighbor on the path P), and all the edges on P are reliable, it follows that v_{j_2} receives the message (m_0, k) from v_{j_2-1} *before* receiving the message (m_0, i) in round r_{j_2}. This contradicts the assumption that $(v_{j_2}, k) \in \mathcal{I}$, as by property (3) of the definition of I_{j_2}, the node v_{j_2} did not receive the message (m_0, k) from v_{j_2-1} before round r_{j_2}. ◁

This completes the proof of Lemma 15. Finally, we show that all nodes accept the message m_0 during the second phase using Lemma 15. This concludes the proof of Theorem 12.

▷ Claim 19. All nodes accept m_0 within $O(L)$ rounds from the beginning of Phase 2.

▶ Remark. Our broadcast algorithm does not need to assume that the nodes know the identity of the source node s. By Corollary 14, in case the adversarial edge e' initiates a false broadcast execution, no node will accept any of the messages sent.

Moreover, the same round complexity also holds in the case where there are multiple sources holding the same broadcast message m_0. This fact will play a role in the final broadcast algorithm where the nodes do not have an estimate on the diameter D.

2.2 Broadcast without Knowing the Diameter

We next show how to remove the assumption that the nodes know an estimate of the diameter; consequently, our broadcast algorithm also computes a linear estimate of the diameter. This increases the round complexity by a logarithmic factor and establishes Theorem 4.
We first describe the algorithm under the simultaneous wake-up assumption and then explain how to remove it.

Algorithm Broadcast. The algorithm applies Algorithm BroadcastKnownDiam of Section 2 for $k = O(\log D)$ iterations in the following manner. Every iteration $i \in \{1, \ldots, k\}$ consists of two steps. In the first step, the source node s initiates BroadcastKnownDiam(D_i) with diameter estimate $D_i = 2^i$, and the desired message m_0. Denote all nodes that accepted the message m_0 by A_i and let $N_i = V \setminus A_i$ be the nodes that did not accept the message.

In the second step, the nodes in N_i inform s that the computation is not yet complete in the following manner. All nodes in N_i broadcast *the same* designated message M by applying Algorithm BroadcastKnownDiam$(9D_i)$ with diameter estimate $9D_i$ and the message M. The second phase can be viewed as performing a single broadcast from $|N_i|$ multiple sources. If the source node s receives and accepts the message M during the second step, it continues to the next iteration $i+1$. If after $\widetilde{O}(D_i^2)$ rounds s did not receive and accept the message M, it broadcasts a termination message M_t to all nodes in V using $\mathsf{BroadcastKnownDiam}(7D_i)$ (with diameter estimate $7D_i$). Once a node $v \in V$ accepts the termination message M_t, it completes the execution with the output message it has accepted so far. Additionally, for an iteration i in which v accepted the termination message, D_i can be considered as an estimation of the graph diameter.

Analysis. We begin with noting that no node $v \in V$ accepts a wrong message $m' \neq m_0$ as its output. This follows by Claim 13 and the correctness of Algorithm BroadcastKnownDiam.

▶ **Observation 20.** *No node $v \in V$ accepts a wrong message $m' \neq m_0$.*

Fix an iteration i. Our next goal is to show that if $N_i \neq \emptyset$, then s will accept the message M by the end of the iteration. Consider the second step of the algorithm where the nodes in N_i broadcast the message M toward s using $\mathsf{BroadcastKnownDiam}(9D_i)$. Since all nodes in N_i broadcast *the same* message M, we refer to the second step as a *single* execution of $\mathsf{BroadcastKnownDiam}(9D_i)$ with multiple sources. We begin with showing that the distance between the nodes in A_i and s is at most $14D_i$.

▷ Claim 21. For every node $v \in A_i$, it holds that $\mathrm{dist}(s, v, G \setminus \{e'\}) \leq 14D_i$.

Proof. Recall that Algorithm BroadcastKnownDiam proceeds in two phases. In the first phase, the source node propagates messages of the form (M, k), and in the second phase, the source node propagates *accept* messages. For a node v that accepts the message m_0 in the i'th iteration, according to Algorithm $\mathsf{BroadcastKnownDiam}(D_i)$, it receives an $\mathit{accept}(m_0)$ message from a neighbor w in Phase 2, and stored a message (m_0, k) in Phase 1, such that $(v, w) \notin G_k$. Let P_1 be the path on which the message $\mathit{accept}(m_0)$ propagated toward v in Phase 2 of $\mathsf{BroadcastKnownDiam}(D_i)$. Since the second phase is executed for $7D_i$ rounds, it holds that $|P_1| \leq 7D_i$. In the case where $e' \notin P_1$, since s is the only node initiating $\mathit{accept}(m_0)$ messages (except maybe e'), P_1 is a path from s to v in $G \setminus \{e'\}$ as required.

Assume that $e' \in P_1$, and denote it as $e' = (v_1, v_2)$. Without loss of generality, assume that on the path P_1, the node v_1 is closer to v than v_2. Hence, v_1 received an $\mathit{accept}(m_0)$ message from v_2 during Phase 2, and because v_1 also sent the message over P_1, it accepted m_0 as its output. Therefore, during the execution of $\mathsf{BroadcastKnownDiam}(D_i)$, the node v_1 stored a message (m_0, j) during the first phase, where $e' \notin G_j$. As all edges in G_j are reliable, we conclude that G_j contains a s-v_1 path P of length $\eta \leq 7D_i$ such that $e' \notin P$. Thus, the concatenated path $P \circ P_1[v_1, v]$ is a path of length at most $14D_i$ from s to v in $G \setminus \{e'\}$ as required. ◁

We now show that if $N_i \neq \emptyset$ then s accepts the message M during the second step and continues to the next iteration. The proof is very similar to the proof of Claim 19 and follows from the following observation.

▶ **Observation 22.** *For every $u \in A_i$ and an edge $e = (v, u)$, it holds that $\mathrm{dist}_{G \setminus \{e', e\}}(N_i, u) \leq 7 \cdot 9D_i$.*

Proof. Let T be a BFS tree rooted at s in $G \setminus \{e'\}$ restricted[3] to the nodes in A_i. By Claim 21 the depth of T is at most $14D_i$. In follows that the forest $T \setminus \{e\}$ contains at most

[3] The tree T might also contain internal nodes in N_i, but it is require to span only the nodes in A_i.

2 trees of diameter $2 \cdot 14D_i$. Since G is 3 edge-connected, there exists a path from some node in N_i to u in $G \setminus \{e, e'\}$.

Hence, the shortest path from N_i to u in $G \setminus \{e, e'\}$ denoted as P can be transformed into a path P' containing at most two edges of P as well as two tree subpaths of the forest $T \setminus \{e\}$. Therefore, $\text{dist}_{G\setminus\{e,e'\}}(N_i, u) \leq |P'| \leq 4 \cdot 14D_i + 2 \leq 7 \cdot 9D_i$. ◀

▷ Claim 23. If $N_i \neq \emptyset$, s accepts the message M by the end of Step 2 of the i'th iteration.

Proof. Let $P = (u_0, u_1, \ldots, u_\eta = s)$ be a shortest path from some node $u_0 \in N_i$ to the source node s. As P is the shortest such path, for every $j \neq 0$ $u_j \in A_i$, and due to Claim 21 $|P| \leq 14D_i + 1$. In order to prove Claim 23 we will show that every node $u_j \in P$ accepts the message M by round j of Phase 2 in the execution of BroadcastKnownDiam$(9D_i)$ (in Step 2), by induction on j.

Base case: as $u_0 \in N_i$, it accepts the message M at the beginning of the phase. Assume the claim holds for u_j and consider the node u_{j+1}. By Observation 22 there exists a path P_{j+1} from some node in N_i to u_{j+1} in $G \setminus \{e', (u_j, u_{j+1})\}$ of length $|P_{j+1}| \leq 7 \cdot 9D_i$. Hence, by Lemma 15 combined with the covering property of the covering subgraphs family used in BroadcastKnownDiam$(9D_i)$, we conclude that u_{j+1} stored a message (M, τ) in Phase 1 where $(u_j, u_{j+1}) \notin G_\tau$ for some neighbor $w \in N(u_{j+1})$.

By the induction assumption, u_j sends u_{j+1} an *accept*(M) message by round j of Phase 2, and since $(u_{j+1}, u_j) \notin G_\tau$, it follow that the node u_{j+1} accepts the message M by round $j + 1$ as required. ◁

Recall that since the second phase of BroadcastKnownDiam(D_i) is executed for $7D_i$ rounds. Hence, when $D_i < D/7$ there must exist a node $w \in V$ that did not accept the message m_0 during the execution of BroadcastKnownDiam(D_i) in the first step, and therefore $N_i \neq \emptyset$. On the other hand, when $D_i \geq D$, all nodes in V receive and accept m_0 during the first step of the i'th iteration and therefore $N_i = \emptyset$. Hence, for an iteration i^* in which no node broadcasts the message M (and therefore s decides to terminate the execution), it holds that $D_{i^*} \in [D/7, D]$. Since s broadcasts the termination message M_t by applying Algorithm BroadcastKnownDiam$(7D_{i^*})$ with diameter estimate $7D_{i^*}$, we conclude that all nodes in V will finish the execution as required. To omit the wake-up assumption, in the second step of each iteration, the broadcast of message M is initiated by nodes in N_i with neighbors in A_i.

3 Broadcast against t Adversarial Edges

In this section, we consider the broadcast problem against t adversarial edges and prove Theorem 5. The adversarial edges are fixed throughout the execution but are unknown to any of the nodes. Given a D–diameter, $(2t + 1)$ edge-connected graph G, and at most t adversarial edges $F \subseteq E$, the goal is for a source node s to deliver a message m_0 to all nodes in the graph. At the end of the algorithm, each node is required to output the message m_0. Our algorithm is again based on a locally known family $\mathcal{G}$ with several desired properties. The algorithm floods the messages over the subgraphs of $\mathcal{G}$. The messages exchanged over each subgraph $G_i \in \mathcal{G}$ contains also the path information along which the message has been received. As we will see, the round complexity of the algorithm is mostly dominated by the cardinality of $\mathcal{G}$.

We use the following fact from [36], whose proof follows by the proof of Fact 11.

▶ **Fact 24** (Implicit in [36]). *Given a graph G and integer parameters L and k, there exists a 0-round algorithm that allows all nodes to locally know an (L, k) covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ such that $\ell = ((Lk \log n)^{k+1})$.*

Towards proving Theorem 5, we prove the following which will become useful also for the improved algorithms for expander graphs.

▶ **Theorem 25.** *Given a $(2t+1)$ edge-connected graph G of diameter D, and a parameter L satisfying that for every $u, v \in V$, and every set $E \subseteq E$ of size $|E| \leq 2t$, it holds that* $\text{dist}_{G \setminus E}(u, v) \leq L$. *Then assuming that the nodes locally know an $(L, 2t)$ covering family $\mathcal{G}$, there exists a deterministic broadcast algorithm* $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}, L, t)$ *against at most t adversarial edges F with round complexity $O(L \cdot |\mathcal{G}|)$.*

We note that by Observation 9, every $(2t+1)$ edge-connected graph G with diameter D satisfies the promise of Theorem 25 for $L = (6t+2)D$. Finally, our algorithm makes use also of the following definition for a minimum s-v cut defined over a collection of s-v *paths*.

▶ **Definition 26** (Minimum (Edge) Cut of a Path Collection)**.** *Given a collection of s-v paths $\mathcal{P}$, the minimum s-v cut in $\mathcal{P}$, denoted as $MinCut(s, v, \mathcal{P})$, is the minimal number of edges appearing on all the paths in $\mathcal{P}$. I.e., letting $MinCut(s, v, \mathcal{P}) = x$ implies that there exists a collection of x edges E' such that for every path $P \in \mathcal{P}$, it holds that $E' \cap P \neq \emptyset$.*

We are now ready to describe the broadcast algorithm given that the nodes know an $(L, 2t)$ covering family $\mathcal{G}$ (along with the parameters L and t) as specified by Theorem 25. Later, we explain the general algorithm that omits this assumption.

Broadcast Algorithm $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}, L, t)$. Similarly to the single adversarial edge case, the algorithm has two phases, a flooding phase and an acceptance phase. In the first phase of the algorithm, the nodes exchange messages over the subgraphs of $\mathcal{G}$, that contain also the path information, along which the bit $\{0, 1\}$ is received. In addition, instead of propagating the messages of distinct G_i's subgraphs in a pipeline manner, we run the entire i'th algorithm (over the edges of the graph G_i) after finishing the application of the $(i-1)$ algorithm[4].
In the first phase, the nodes flood *heard bundles* over all the $G_i \in \mathcal{G}$ subgraphs, defined as follows.

Heard bundles. A *bundle* of heard messages sent from node v to u consists of:

1. A header message $heard(m, len, P)$, where P is an s-v path of length len along which v received the message m.
2. A sequence of len messages specifying the edges of P, one by one.

This bundle contains $len + 1$ messages that will be sent in a pipeline manner in the following way. The first message is the header $heard(m, len, P)$ sent in round τ. Then in the next consecutive len rounds, v sends the edges of P in reverse order (from the edge incident to v to s).

Phase 1: Flooding. The first phase consists of $\ell = |\mathcal{G}|$ iterations, where each iteration is implemented using $O(L)$ rounds. At the first round of the i'th iteration, the source node s sends the message $heard(m_0, 1, \emptyset)$ to all neighbors. Every node v, upon receiving the *first* bundle message $heard(m', x, P)$ over an edge in G_i from a neighbor w, stores the bundle $heard(m', x+1, P \cup \{w\})$ and sends it to all neighbors. Note that each node stores and sends at most *one* heard bundle $heard(m', x, P)$ in each iteration (and not one per message m').

[4] One might optimize the $O(t)$ exponent by employing a pipeline approach in this case as well.

Phase 2: Acceptance. The second phase consists of $O(L)$ rounds, in which *accept* messages are propagated from the source s to all nodes as follows. In the first round s sends $accept(m_0)$, to all neighbors. Every node $v \in V \setminus \{s\}$ decides to accept a message m' if the following two conditions hold: (i) v receives $accept(m')$ from a neighbor w, and (ii) $\mathrm{MinCut}(s, v, \mathcal{P}) \geq t$, where

$$\mathcal{P} = \{P \mid v \text{ stored a } heard(m', len, P) \text{ message and } (v, w) \notin P\} \ . \qquad (1)$$

Note that since the decision here is made by computing the minimum cut of a path collection, it is indeed required (by this algorithm) to send the path information.

Correctness. We begin with showing that no node accepts a false message.

▷ Claim 27. No node $v \in V$ accepts a message $m' \neq m_0$ in the second phase.

Proof. Assume by contradiction there exists a node that accepts a false message m', and let v be the *first* such node. By first we mean that any other node that accepted m' accepted the message in a later round than v breaking ties arbitrarily. Hence, v received a message $accept(m')$ from some neighbor w. Because v is the first such node, the edge (w, v) is adversarial. Let $E' = F \setminus \{(w, v)\}$ be the set of the remaining $t - 1$ adversarial edges, and let $\mathcal{P}$ be given as by Eq. (1). We next claim that $\mathrm{MinCut}(s, v, \mathcal{P}) \leq t - 1$ and thus v does not accept m'.

To see this, observe that any path P such that v received a message $heard(m', len, P)$ must contain at least one edge in E'. This holds even if the content of the path P is corrupted by the adversarial edges. Since there are at most $t - 1$ edges in E' and all the paths in $\mathcal{P}$ are passing through them, it holds that $\mathrm{MinCut}(s, v, \mathcal{P}) \leq t - 1$ as required. ◁

Finally, we show that all nodes in V accept the message m_0 during the second phase. This completes the proof of Theorem 25.

▷ Claim 28. All nodes accept m_0 within $O(L)$ rounds from the beginning of Phase 2.

Proof. We will show that all nodes accept the message m_0 by induction on the distance from the source s in the graph $G \setminus F$. Let T be some BFS tree rooted at s in $G \setminus F$. The base case holds vacuously, as s accepts the message m_0 in round 0. Assume all nodes at distance at most i from s in $G \setminus F$ accepted the message by round i. Consider a node v at distance $i + 1$ from s in $G \setminus F$. By the induction assumption on layer i, v receives the message $accept(m_0)$ from a neighbor w in round $j \leq i$ over a reliable edge (w, v). We are left to show that $\mathrm{MinCut}(s, v, \mathcal{P}) \geq t$, where $\mathcal{P}$ is as given by Eq. (1). Alternatively, we show that for every edge set $E' \subseteq E \setminus \{(w, v)\}$ of size $t - 1$, the node v stores a heard bundle containing m_0 and a path P_k such that $P_k \cap (E' \cup \{(v, w)\}) = \emptyset$ during the first phase. This necessary implies that the minimum cut is at least t.

For a subset $E' \subseteq E$ of size $t - 1$, as $|F \cup E' \cup \{w, v\}| \leq 2t$, by the promise on L in Theorem 25, $\mathrm{dist}_{G \setminus (F \cup E' \cup \{w,v\})}(s, v) \leq L$. By the covering property of the covering family $\mathcal{G}$ it follows that there exists a subgraph G_k such that $G_k \cap (F \cup E' \cup \{(v, w)\}) = \emptyset$, and $\mathrm{dist}_{G_k}(s, v) \leq L$. Hence, all edges in G_k are reliable, and the only message passed through the heard bundles during the k'th iteration is the correct message m_0. Additionally, as $\mathrm{dist}_{G_k}(s, v) \leq L$, the node v stores a heard bundle $heard(m_0, x, P_k)$ during the k'th iteration, for some s-v path P_k of length $x = O(L)$. As $P_k \subseteq G_k$ it also holds that $P_k \cap (E' \cup \{(v, w)\}) = \emptyset$. We conclude that $\mathrm{MinCut}(s, v, \mathcal{P}) \geq t$, and by the definition of Phase 2, v accepts m_0 by round $j + 1 \leq i + 1$. The claim follows as the diameter of T is $O(L)$ by the promise on L. ◁

Algorithm Broadcast (Proof of Theorem 5). We now describe the general broadcast algorithm. Our goal is to apply Algorithm $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}, L, t)$ over the $(L, 2t)$ covering family $\mathcal{G}$ for $L = O(tD)$, constructed using Fact 24. Since the nodes do not know the diameter D (or a linear estimate of it), we make $O(\log D)$ applications of Algorithm $\mathsf{BroadcastKnownCovFamily}(\widehat{\mathcal{G}}, \widehat{L}, t)$ using the $(\widehat{L}, 2t)$ covering family $\widehat{\mathcal{G}}$, for $\widehat{L} = O(t\widehat{D})$ where $\widehat{D} = 2^i$ is the diameter guess for the i'th application.

Specifically, at the beginning of the i'th application, the source s initiates Algorithm $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}_i, L_i, t)$ with the desired message m_0 over the $(L_i, 2t)$ covering family $\mathcal{G}_i$ constructed using Fact 24 with $L_i = O(tD_i)$ and $D_i = 2^i$. Denote all nodes that accepted the message m_0 at the end of Algorithm $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}_i, L_i, t)$ by A_i, and let $N_i = V \setminus A_i$ be the nodes that did not accept the message.

The algorithm now applies an additional step where the nodes in N_i inform s that they did not accept any message in the following manner. All nodes in N_i broadcast *the same* designated message M, by applying Algorithm $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}'_i, ctL_i, t)$ over an $(ctL_i, 2t)$ covering family $\mathcal{G}'_i$, for some fixed constant $c > 0$ (known to all nodes). This can be viewed as performing a single broadcast execution (i.e., with the same source message) but from $|N_i|$ multiple sources. We next set $\tau_i = O(t \cdot D_i \log n)^{O(t)}$ as a bound on the waiting time for a node to receive any acknowledgment.

If the source node s accepts the message M at the end of this broadcast execution, it waits τ_i rounds, and then continues to the next application[5] $i + 1$ (with diameter guess 2^{i+1}). In the case where s did not accept the message M within τ_i rounds from the beginning of that broadcast execution, it broadcasts a termination message M_T to all nodes in V. This is done by applying Algorithm $\mathsf{BroadcastKnownCovFamily}(\mathcal{G}'_i, ctL_i, t)$ over the $(ctL_i, 2t)$ covering family $\mathcal{G}'_i$. Once a node $v \in V$ accepts the termination message M_T, it completes the execution with the last message it has accepted so far (in the analysis part, we show that it indeed accepts the right message). A node v that did not receive a termination message M_T within τ_i rounds, continues to the next application of Algorithm $\mathsf{BroadcastKnownCovFamily}$.

The correctness argument exploits the fact that for an application i such that $N_i \neq \emptyset$, the graph G' obtained by contracting [6] all nodes in N_i into a single node a, satisfies the following: (i) it is $(2t + 1)$ edge-connected, (ii) it contains s, and (iii) it has diameter $O(L_i) = O(t \cdot D_i)$. A complete analysis of the algorithm can be found in the full paper [32].

We observe that our broadcast algorithm can be implemented in the LOCAL model using $O(tD \log n)$ many rounds.

▶ **Corollary 29.** *For every $(2t + 1)$ edge-connected graph, and a source node s, there is a deterministic broadcast algorithm against t adversarial edges that runs in $O(tD \log n)$ local rounds.*

Proof. The algorithm is the same as in the $\mathsf{CONGEST}$ model. However, since in the local model there are no bandwidth restrictions, the message propagation over the $|\mathcal{G}|$ subgraphs of the (L, t) covering family can be implemented simultaneously within $L = O(tD)$ rounds. ◀

[5] We make the source node s wait since in the case where it actually sends a termination message, all nodes accept it within τ_i rounds. Therefore, we need to make sure that all nodes start the next $i + 1$ application at the same time.

[6] I.e., we contract all edges with both endpoints in N_i.

References

1 Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 317–326, 2019.

2 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected O(1) rounds, expected o(n^2) communication, and optimal resilience. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, pages 320–334, 2019.

3 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

4 John Augustine, Gopal Pandurangan, and Peter Robinson. Fast byzantine leader election in dynamic networks. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 276–291, 2015.

5 Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

6 Anindo Bagchi and S. Louis Hakimi. Information dissemination in distributed systems with faulty units. *IEEE Transactions on Computers*, 43(6):698–710, 1994.

7 Piotr Berman and Juan A. Garay. Cloture votes: n/4-resilient distributed consensus in t+1 rounds. *Math. Syst. Theory*, 26(1):3–19, 1993.

8 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415. IEEE Computer Society, 1989.

9 Douglas M Blough and Andrzej Pelc. Optimal communication in networks with randomly distributed byzantine faults. *Networks*, 23(8):691–701, 1993.

10 Greg Bodwin, Michael Dinitz, and Caleb Robelle. Optimal vertex fault-tolerant spanners in polynomial time. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2924–2938. SIAM, 2021.

11 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

12 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

13 Keren Censor-Hillel and Tariq Toukan. On fast and robust information spreading in the vertex-congest model. *Theoretical Computer Science*, 2017.

14 Diptarka Chakraborty and Keerti Choudhary. New extremal bounds for reachability and strong-connectivity preservers under failures. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, pages 25:1–25:20, 2020.

15 Bogdan S. Chlebus, Dariusz R. Kowalski, and Jan Olkowski. Fast agreement in networks with byzantine nodes. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 30:1–30:18, 2020.

16 Brian A. Coan and Jennifer L. Welch. Modular construction of a byzantine agreement protocol with optimal message bit complexity. *Inf. Comput.*, 97(1):61–85, 1992.

17 Ran Cohen, Iftach Haitner, Nikolaos Makriyannis, Matan Orland, and Alex Samorodnitsky. On the round complexity of randomized byzantine agreement. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, pages 12:1–12:17, 2019.

18 Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 169–178. ACM, 2011.

19 Danny Dolev. The byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982.

20 Danny Dolev, Michael J. Fischer, Robert J. Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

21 Danny Dolev and Ezra N. Hoch. Constant-space localized byzantine consensus. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 167–181, 2008.

22 Cynthia Dwork, David Peleg, Nicholas Pippenger, and Eli Upfal. Fault tolerance in networks of bounded degree. *SIAM J. Comput.*, 17(5):975–988, 1988.

23 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

24 Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International Conference on Fundamentals of Computation Theory*, pages 127–140. Springer, 1983.

25 Mattias Fitzi and Ueli Maurer. From partial consistency to global broadcast. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 494–503, 2000.

26 Chaya Ganesh and Arpita Patra. Broadcast extensions with optimal communication and round complexity. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 371–380, 2016.

27 Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for $n >$ 3t processors in $t + 1$ rounds. *SIAM J. Comput.*, 27(1):247–290, 1998.

28 Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 3–12, 2015.

29 Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil P. Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 120–129, 2012.

30 Fabrizio Grandoni and Virginia Vassilevska Williams. Faster replacement paths and distance sensitivity oracles. *ACM Transactions on Algorithms (TALG)*, 16(1):1–25, 2019.

31 Fabrizio Grandoni and Virginia Vassilevska Williams. Faster replacement paths and distance sensitivity oracles. *ACM Trans. Algorithms*, 16(1):15:1–15:25, 2020.

32 Yael Hitron and Merav Parter. Broadcast CONGEST algorithms against adversarial edges. *CoRR*, abs/2004.06436, 2021.

33 Yael Hitron and Merav Parter. General CONGEST algorithms compilers against adversarial edges. In *DISC*, 2021.

34 Damien Imbs and Michel Raynal. Simple and efficient reliable broadcast in the presence of byzantine processes. *arXiv preprint arXiv:1510.06882*, 2015.

35 David R Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

36 Karthik C. S. and Merav Parter. Deterministic replacement path covering. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 704–723, 2021.

37 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

38 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact byzantine consensus on undirected graphs under local broadcast model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 327–336, 2019.

39 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 87–98, 2006.

40 Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 275–282. ACM, 2004.

41 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

42 Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

43 Alexandre Maurer and Sébastien Tixeuil. On byzantine broadcast in loosely connected networks. In *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, pages 253–266, 2012.

44 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, pages 28:1–28:17, 2020.

45 Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. Comput. Syst. Sci.*, 49(2):149–167, 1994.

46 Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

47 Merav Parter and Eylon Yogev. Congested clique algorithms for graph spanners. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

48 Merav Parter and Eylon Yogev. Distributed algorithms made secure: A graph theoretic approach. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1693–1710. SIAM, 2019.

49 Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 107–116, 2019.

50 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

51 Andrzej Pelc. Fault-tolerant broadcasting and gossiping in communication networks. *Networks: An International Journal*, 28(3):143–156, 1996.

52 Andrzej Pelc and David Peleg. Broadcasting with locally bounded byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005.

53 Andrzej Pelc and David Peleg. Feasibility and complexity of broadcasting with random transmission failures. *Theoretical Computer Science*, 370(1-3):279–292, 2007.

54 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* SIAM, 2000.

55 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 304–313. Springer, 1989.

56 Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In *Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings*, pages 358–367, 1990.

57 Sam Toueg, Kenneth J Perry, and TK Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16(3):445–457, 1987.

58 Eli Upfal. Tolerating a linear number of faults in networks of bounded degree. *Inf. Comput.*, 115(2):312–320, 1994.

59 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

60 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):1–13, 2013.

61 Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1130–1143, 2017.

General CONGEST Compilers against Adversarial Edges

Yael Hitron
Weizmann Institute of Science, Rehovot, Israel

Merav Parter
Weizmann Institute of Science, Rehovot, Israel

Abstract

We consider the *adversarial* CONGEST model of distributed computing in which a fixed number of edges (or nodes) in the graph are controlled by a computationally unbounded adversary that corrupts the computation by sending malicious messages over these (a-priori unknown) controlled edges. As in the standard CONGEST model, communication is synchronous, where per round each processor can send $O(\log n)$ bits to each of its neighbors.

This paper is concerned with distributed algorithms that are both time efficient (in terms of the number of rounds), as well as, robust against a fixed number of adversarial edges. Unfortunately, the existing algorithms in this setting usually assume that the communication graph is complete (n-clique), and very little is known for graphs with arbitrary topologies. We fill in this gap by extending the methodology of [Parter and Yogev, SODA 2019] and provide a compiler that simulates any CONGEST algorithm $\mathcal{A}$ (in the reliable setting) into an equivalent algorithm $\mathcal{A}'$ in the adversarial CONGEST model. Specifically, we show the following for every $(2f+1)$ edge-connected graph of diameter D:

- For $f = 1$, there is a general compiler against a single adversarial edge with a compilation overhead of $\widehat{O}(D^3)$ rounds[1]. This improves upon the $\widehat{O}(D^5)$ round overhead of [Parter and Yogev, SODA 2019] and omits their assumption regarding a fault-free preprocessing phase.
- For any constant f, there is a general compiler against f adversarial edges with a compilation overhead of $\widehat{O}(D^{O(f)})$ rounds. The prior compilers of [Parter and Yogev, SODA 2019] were limited to a single adversarial edge.

Our compilers are based on a new notion of fault-tolerant cycle covers. The computation of these cycles in the adversarial CONGEST model constitutes the key technical contribution of the paper.

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases CONGEST, Cycle Covers, Byzantine Adversaries

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.24

Related Version *Full Version*: https://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/Adversarial-CONGEST-Compilers.pdf

Funding *Merav Parter*: This project is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949083).

Acknowledgements We are very grateful to Dan Mikulincer, David Peleg and Eylon Yogev for many useful discussions. More specifically, we thank Eylon Yogev for suggesting the high-level framework for the 1-FT cycle cover. We also thank the unanimous referee for the useful comments on the proof of Lemma 21.

1 Introduction

As communication networks grow in size, they become increasingly vulnerable to failures and byzantine attacks. It is therefore crucial to develop fault-tolerant distributed algorithms that work correctly despite the existence of such failures, without knowing their location. The

[1] The notation $\widehat{O}(.)$ hides factors of $2^{O(\sqrt{\log n})}$ which arises by the distributed algorithms of [34, 35].

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 24; pp. 24:1–24:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

area of fault-tolerant distributed computation has attracted a lot of attention over the years, especially since the introduction of the byzantine agreement problem by [Pease, Shostak and Lamport, JACM'80] [37]. The vast majority of these algorithms, however, assume that the communication graph is the complete graph [12, 13, 16, 8, 43, 41, 6, 42, 5, 15, 18, 17, 27, 38, 25, 14, 29, 23, 10, 26]. For the latter, one can provide time efficient algorithms for various distributed tasks that can tolerate up to a *constant* fraction of corrupted edges and nodes [12, 4, 6, 14]. Very little is known on the complexity of fault-tolerant computation for general graph topologies. In a seminal work, Dolev [12] showed that any given graph can tolerate up to f adversarial nodes iff it is $(2f + 1)$ vertex-connected. Unfortunately, the existing distributed algorithms for general $(2f + 1)$ connected graphs, usually require a polynomial number of rounds in the CONGEST model of distributed computing [39].

In this paper, we present a general compiler that translates any given distributed algorithm $\mathcal{A}$ (in the fault-free setting) into an equivalent algorithm $\mathcal{A}'$ that performs the same computation in the presence of f adversarial edges. Our primary objective is to minimize the **compilation overhead**, namely, the ratio between[2] the round complexities of the algorithms $\mathcal{A}'$ and $\mathcal{A}$. We take the gradual approach of fault-tolerant network design, and consider first the case of a single adversarial edge, and later on the case of multiple adversarial edges. We note that, in general, such compilers might not be obtained for adversarial nodes[3] and thus we focus on edges.

1.1 Model Definition and the State of the Art

Very recently, [21] presented the first round-efficient broadcast algorithms against adversarial edges in the CONGEST model. [21] also formalized the adversarial CONGEST model, which is the model that we consider in this work as well.

The Adversarial CONGEST Model. The network is abstracted as an n-node graph $G = (V, E)$, with one processor on each node. Each node has a unique identifier of $O(\log n)$ bits. Initially, the processors only know the identifiers of their incident edges[4], as well as a polynomial estimate on the number of nodes n.

There is a computationally unbounded adversary that controls a fixed set of edges F^* in the graph. The set of F^* edges are denoted as *adversarial*, and the remaining edges $E \setminus F^*$ are denoted as *reliable*. The nodes do not know the identity of the adversarial edges in F^*, but they do know the bound f on the cardinality of F^*. We consider the full information model where the adversary knows the graph, the messages sent through the graph edges in each round, and the internal randomness and the input of the nodes. On each round, the adversary can send $O(\log n)$ bits along each of the edges in F^*. The adversary is adaptive as it can determine its behavior in round r based on the overall communication up to round r.

We focus on $(2f + 1)$ edge-connected graphs, which can tolerate up to f adversarial edges. The problem of devising general round-by-round compilers in the adversarial CONGEST model boils down into the following distributed task:

[2] Note that we use the term compilation overhead to measure the time it takes to simulate a single fault-free round of algorithm $\mathcal{A}$ in the adversarial setting. This should not be confused with the time required to set up the compiler machinery (e.g., computing the cycle cover).

[3] Such compilers might still be obtained under the stronger KT2 model where nodes know their two-hop neighbors.

[4] This is known as the standard KT1 model [3].

Single Round Compilation in the Adversarial CONGEST Model: Given is a $(2f+1)$ edge-connected graph $G=(V,E)$ with a fixed set $F^* \subseteq G$ of at most f adversarial edges. Let $\mathcal{M} = \{M_{u \to v} \mid (u,v) \in E\}$ be a collection of $O(\log n)$-bit messages that are required to be sent over (potentially) all graph edges. I.e., for each (directed) edge (u,v), the node u has a designated $O(\log n)$-bit message for v.

The single round compilation algorithm is required to exchange these messages in the adversarial CONGEST model, such that at the end of the algorithm, each node v holds the correct message $M_{u \to v}$ for each of its neighbors u, while ignoring all remaining (corrupted) messages.

The main complexity measure is the round complexity of the single-round compilation algorithm which corresponds to the *compilation overhead* of the compiler. The compilation of CONGEST algorithms under various adversarial settings, has been recently studied by [33]. We next explain their methodology and discuss our contribution with respect to the state-of-the-art.

The simulation methodology of [33]. Motivated by various applications for *resilient distributed computing*, Parter and Yogev [33] introduced the notion of *low-congestion cycle covers* as a basic communication backbone for reliable communication. Formally, a (c,d)-cycle cover of a two edge-connected graph G is a collection of cycles in G in which each cycle is of length at most d, and each edge participates in at least one cycle and at most c cycles. The quality of the cycle cover is measured by $c+d$. Using the beautiful result of Leighton, Maggs and Rao [28] and the follow-up by Ghaffari [19], a (c,d)-cycle cover allows one to route $O(\log n)$ bits of information on all cycles simultaneously in $\widetilde{O}(c+d)$ CONGEST rounds.

Low-congestion cycle covers with parameters c,d give raise to a *simulation* methodology that transforms any distributed algorithm $\mathcal{A}$ and compile it into a *resilient* one; the compilation overhead is $g(c,d)$, for some function g. The resilient simulation exploits the fact that a cycle covering an edge $e=(u,v)$ provides *two*-edge-disjoint paths for exchanging messages from u to v. Parter and Yogev [33] showed that any n-node two edge-connected graph with diameter D has a (c,d)-cycle covers with $c=O(1)$ and $d=\widetilde{O}(D)$. These bounds are existentially tight. [34, 35] also presented an r-round CONGEST algorithm for computing (c,d) cycles covers for $r,d=\widehat{O}(D)$ and $c=\widehat{O}(1)$.

Our simulation methodology in the adversarial CONGEST model extends the work of [33] in several aspects. First, the cycle covers of [33] are limited to handle at most *one* edge corruption. To accommodate a large number of adversarial edges, we introduce the notion of *fault-tolerant (FT) cycle covers* which extends low-congestion cycle cover to handle multiple adversarial edges. Informally, a FT cycle cover with parameters c,d is a cycle collection $\mathcal{C}$ that covers each edge e by multiple cycles (instead of one): For every sequence of at most f faults F, there is a cycle C in $\mathcal{C}$ that covers[5] e without visiting any of the edges in $F \setminus \{e\}$. All cycles in $\mathcal{C}$ are required to be of length at most d, and with an overlap of at most c, to allow an efficient information exchange over all these cycles in parallel.

A key limitation of the compilers provided by [33] is that they assume the cycle covers are computed in a (fault-free) preprocessing phase. These cycles are then used by the compilers in the adversarial CONGEST model. Our main goal in this paper is to omit this assumption and provide efficient algorithms for computing the FT cycle covers in

[5] A stricter requirement is to cover each edge by f edge-disjoint cycles, however, this definition leads to a larger compilation overhead compared to the one obtained with our definition.

the adversarial CONGEST model. The computation of these cycles in the presence of the adversarial edges is quite intricate. The key challenge is in computing cycles for covering the adversarial edges themselves. The latter task requires some coordination between the endpoints of the adversarial edges, which seems to be quite hard to achieve. Note that the covering of the adversarial edges by cycles is indeed crucial for the compilation task, in order to reliably simulate the message exchange over these edges in the given fault-free algorithm. Upon computing FT cycle covers with parameter c, d, we then present a round-by-round compiler whose overhead depends on the c, d parameters. To optimize for the round overhead, we exploit (our modified) FT cycle covers in a somewhat more delicate manner compared to that of [33], leading to an improvement by factor of $O(D^2)$ rounds.

1.2 Contributions and Key Results

We consider the design of compilers that can simulate every given distributed algorithm in the adversarial CONGEST model. The compilers are based on a new notion of *FT cycle cover*, an extension of the low-congestion cycle cover [31] to the adversarial setting. We also provide a new method to compile the algorithm given the FT cycle cover. We start by describing our contribution w.r.t the combinatorial characterization of FT cycle covers, and then turn to consider the computational aspects in the adversarial CONGEST model.

1.2.1 Combinatorial Properties of Fault Tolerant Cycle Covers

We provide first the standard definition of low congestion cycle covers of [33], and then introduce their extension to the fault-tolerant setting. A (c, d) *low-congestion cycle cover* of a two edge-connected graph G is a collection of cycles in G in which (i) each cycle is of length at most d (dilation), and (ii) each edge participates in at least *one* cycle (covering), and at most c cycles (congestion). The *quality* of the cycle cover is measured by $c + d$. To provide reliable computation in the presence of f adversarial edges F^*, it is desired to cover each edge by multiple short cycles with small overlap. This motivates the following definition.

▶ **Definition 1** (f-FT Cycle Covers). *Given an $(f + 1)$ edge-connected graph G an f-FT cycle cover with parameters (c, d) is a collection of cycles $\mathcal{C}$ such that for any set $E' \subseteq E$ of size $(f - 1)$ and every edge $e \in E$, there exists a cycle $C \in \mathcal{C}$ such that $C \cap (E' \cup \{e\}) = \{e\}$. The length of every cycle in $\mathcal{C}$ is at most d, and each edge participates in at most c cycles.*

In other words, the f-FT cycle cover $\mathcal{C}$ provides for each edge $e = (u, v)$ a subgraph G'_e (consisting of all cycles covering e), such that the minimum u-v cut in G'_e is at least $f + 1$. Using the FT sampling technique from [44, 11], in the full version we show the following:

▶ **Lemma 2** (Upper bound on FT Cycle-Covers). *For every $(f + 1)$ edge-connected graph G with diameter D, there is a randomized construction for computing f-FT cycle cover $\mathcal{C}$ with parameters (c, d) where $c = f(5fD)^f \cdot \mathrm{poly}(\log n)$ and $d = 5fD$.*

One of our technical contributions is an almost *matching* lower bound for the quality of FT cycle covers. This is done by a careful analysis of the congestion and dilation parameters of replacement paths in faulty graphs. We believe that the following graph theoretical theorem should be of independent interest in the context of fault-tolerant network design and distributed minimum cut computation.

▶ **Theorem 3** (Lower Bound on the Quality of FT Cycle Covers). *For every $f \geq 1$, $D \geq f$ and $n = \omega(D^f)$, there exists an n-node $(f + 1)$ edge-connected graph $G^* = (V, E)$ with diameter D, such that any f-FT cycle cover with parameters c, d must satisfy that $c + d = (D/f)^{\Omega(f)}$.*

This theorem provides an explanation for the compilation overhead of $D^{O(f)}$ of our compilers. It also provides an explanation for the natural barrier of $D^{O(f)}$ rounds for handling f adversarial edges in the distribued setting. Specifically, the lower bound implies that there exists at least one pair of nodes u, v in the graph G^* such that for any selection of $f+1$ edge-disjoint u-v paths $\mathcal{P}$ in G^*, the longest path in $\mathcal{P}$ must have length of $(D/f)^{\Omega(f)}$ edges. Theorem 3 also proves that the collection of all $V \times V \times E^f$ replacement paths[6] avoiding f faults, obtained by the FT sampling technique, are optimal in terms of their congestion + dilation bounds. It also shows that the analysis of the distributed minimum cut algorithm of [30] is nearly *optimal*[7].

A relaxed notion of FT cycle covers. In a setting where a fixed set of edges F^* are adversarial for $|F^*| = f$, it might not be possible to compute $(2f)$-FT cycle cover as defined by Definition 1. This is despite the fact that we require the edge connectivity of the graph to be at least $2f+1$. To see this, consider the scenario where the adversarial edges F^* are completely idle throughout the distributed computation. In such a case, the communication graph becomes $G \setminus F^*$, which is no longer guaranteed to have an edge-connectivity of $2f+1$. For this reason, we consider a more relaxed notion of FT cycle covers, that on the one hand can be computed in the adversarial setting, and on the other hand is strong enough for our compilers.

▶ **Definition 4** ((f, F^*)-FT Cycle Cover). *Given an $(2f+1)$ edge-connected graph G, and a fixed set of unknown adversarial edges $F^* \subseteq E$ of size at most f, an (f, F^*)-FT cycle cover with parameters (c, d) is a collection of cycles $\mathcal{C}$ such that for every edge $e \in E$ (possibly $e \in F^*$), and every set $E' \subseteq E$ of size $|E'| \leq f-1$, there exists a cycle $C \in \mathcal{C}$ such that $C \cap (E' \cup F^* \cup \{e\}) = \{e\}$. The length of each cycle is bounded by d, and every edge appears on at most c cycles in $\mathcal{C}$.*

Note that for every $F \subseteq E$, $|F| \leq f$, an (f, F)-FT cycle cover $\mathcal{C}$ *contains* an f-FT cycle cover, and therefore the lower bound of Theorem 3 also holds for (f, F^*)-FT cycle cover. When $F^* = \{e'\}$, we slightly abuse notation and simply write (f, e')-FT cycle covers Our FT cycle covers should be useful for many other adversarial settings. Specifically, they provide an immediate extension of the compilers of [33] to handle adversaries that corrupt multiple edges, such as eavesdroppers [33] and semi-honest adversaries [32].

We next turn to consider the computational aspects of FT cycle covers, and their applications. In the distributed setting, we assume throughout that the nodes of the graph obtain a linear estimate[8] on the diameter of the graph D. This assumption (also applied in e.g., [9]) is needed as the compilation overhead is a function of D.

1.2.2 Handling a Single Adversarial Edge

We start by considering an adversarial setting with a *single* fixed unknown adversarial edge e'. At the heart of the compiler lies an efficient construction of a $(1, e')$ FT cycle cover in the adversarial CONGEST model.

[6] A replacement path is a shortest path in some graph $G \setminus F$.

[7] This algorithm computes the minimum cut by computing for each vertex v the collection of all replacement paths w.r.t a fixed source node s.

[8] This assumption can be omitted using the broadcast algorithms of [21, 22], in the case where the nodes have a designated marked leader.

▶ **Theorem 5** ($(1, e')$-FT Cycle Cover). *Consider a* 3 *edge-connected n-node graph G of diameter D, and a fixed adversarial edge e'.*

- *There is an r-round deterministic algorithm for computing a $(1, e')$-FT cycle cover with congestion and dilation $c = \widehat{O}(D^2), d = \widehat{O}(D)$, and $r = \widehat{O}(D^4)$ in the adversarial* CONGEST *model.*
- *There is an r-round randomized algorithm for computing a $(1, e')$-FT cycle cover, w.h.p., with congestion and dilation $c, d = \widehat{O}(D)$, and $r = \widehat{O}(D^2)$ in the adversarial* CONGEST *model.*

In the distributed output format of the $(1, e')$-FT cycle cover computation, the endpoints of every edge $e = (u, v)$ hold the unique identifiers of all the cycles $\mathcal{C}_e$ covering e ,as well as, their neighbors on each of these cycles. The key challenge in proving Theorem 5 is in covering the adversarial edge e'. For that purpose we provide a delicate cycle verification procedure that allows the endpoints of each edge $e = (u, v)$ to correctly identify if e is currently covered by a (legal) cycle. This verification is robust to the behavior of the adversarial edge. Using these cycle covers, we obtain general compilers against e'.

▶ **Theorem 6** (Compiler against a Single Adversarial Edge). *Given is a* 3 *edge-connected D–diameter graph G with a fixed adversarial edge e', and a $(1, e')$-FT cycle cover $\mathcal{C}$ with parameters (d, c) for G (e.g., as obtained by Theorem 5). Then any distributed algorithm $\mathcal{A}$ can be compiled into an equivalent algorithm $\mathcal{A}'$ against e' with an overhead of $O(c \cdot d^2)$ rounds (in the adversarial* CONGEST *model).*

This improves the compilation overhead of Parter and Yogev [33] by a factor of $\widetilde{O}(D^2)$ rounds. The compilers of [33] are based on exchanging the $M_{u \to v}$ messages of Alg. $\mathcal{A}$ along 3 edge-disjoint u-v paths. In our compilation scheme, instead of insisting on edge-disjoint paths, the messages are exchanged over a collection u-v paths of a sufficiently large *flow*. This leads to improvement in the compilation overhead.

1.2.3 Handling Multiple Adversarial Edges

We next consider $(2f + 1)$ edge-connected graphs of diameter D with a fixed set $F^* \subseteq E$ of adversarial edges, $|F^*| \leq f$. To handle f adversarial edges F^* in $(2f + 1)$ edge-connected graphs, we use the notion of (f, F^*)-FT cycle covers. Our first contribution is the construction of the (f, F^*)-FT cycle covers in the adversarial CONGEST model. Due to technicalities arises in this adversarial setting, our final output contains the desired cycles required by (f, F^*)-FT cycle cover, but might include in addition, also truncated paths which are quite “harmless” in the compilation process later on. Formally, our distributed construction computes (f, F^*)-FT cycle cover* where the asterisk indicates the possible existence of truncated paths in the distributed output.

▶ **Definition 7** ((f, F^*)-FT Cycle Cover*). *Given a $(2f + 1)$ edge-connected graph G and a fixed set of adversarial edges $F^* \subseteq E$ where $|F^*| \leq f$, a (f, F^*)-FT cycle cover* with parameters (c, d) is a collection of cycles and paths $\mathcal{C}$ such that $\mathcal{C}$* contains *a (f, F^*)-FT cycle cover for G. The length of each cycle and path in $\mathcal{C}$ is at most d and every edge $e \in E$ appears in at most c cycles and paths.*

▶ **Theorem 8** ((f, F^*)-FT Cycle Cover*). *Let G be a $(2f + 1)$ edge-connected graph G of diameter D, and a fixed set of f adversarial edges F^*. Then, there exists an r-round deterministic algorithm, in the adversarial* CONGEST *model, for computing a (f, F^*)-FT cycle cover* for G with parameters $d = \widehat{O}(f \cdot D)$ and $r, c = \widehat{O}((Df \log n)^{O(f)})$.*

Note that by the lower bound result of Theorem 3, the quality of the FT cycle covers must be $(D/f)^{\Omega(f)}$. Given a (f, F^*)-FT cycle cover* for a graph G, we extend the general compiler of Theorem 6 to handle f adversarial edges.

▶ **Theorem 9** (Compilers against f Adversarial Edges). *Given a $(2f+1)$ edge-connected D–diameter graph G with a fixed set of f adversarial edges F^*, and a (f, F^*)-FT cycle cover* with parameters (d, c) for G. Then any distributed algorithm $\mathcal{A}$ can be compiled into an equivalent algorithm $\mathcal{A}'$ against F^*, with a compilation overhead of $O(c \cdot d^3)$ rounds.*

The high level intuitive idea of our compiler is as follows. Fix a round i of algorithm $\mathcal{A}$, and consider the message $M_{u \to v}$ sent over the edge (u, v) in that round. Our compiler lets u send the message $M_{u \to v}$ through all cycles covering e in the (f, F^*)-FT cycle cover*. The node v can then recover $M_{u \to v}$ by exploiting the following property. On the one hand, the (f, F^*)-FT cycle cover* covers e by sufficiently many cycles that avoid $F^* \setminus \{e\}$. Consequently, the correct message $M_{u \to v}$ is received by v over a path collection with a u-v flow[9] at least $f+1$. On the other hand, any corrupted message $M' \neq M_{u \to v}$ must be propagated along a walk that contains at least *one* adversarial edge. Consequently, a corrupted message M' is propagated over a walk collection with a u-v flow at most f.

Technical comparison with [21]. The recent work of [21] provides broadcast algorithms in the adversarial CONGEST model. This paper is concerned with a general compiler that translates any CONGEST algorithm into an adversarial CONGEST algorithms provided that the adversary controls at most f edges in the graph. The common tool used by both of the works is the covering family obtained by the FT sampling and its recent derandomization [24, 7]. Besides this, each paper handles different types of challenges. In the broadcast task the goal is to send the broadcast message m_0 through a collection of sufficiently many reliable paths. In contrast, in the compiler setting, given a (fault-free) algorithm $\mathcal{A}$, the goal is to exchange messages of $\mathcal{A}$ over (potentially) *all* graph edges in a reliable manner. Specifically, unlike the broadcast setting, one cannot simply ignore the adversarial edges (e.g., by exchanging messages over a reliable subgraph $G' \subseteq G$), as it is required to exchange messages in a reliable manner over the endpoints of the *adversarial* edges as well. The heart of this simulation is in the computation of fault-tolerant cycle covers.

1.3 Preliminaries

Notations. Throughout, the diameter of the given graph G is denoted by D, and the number of nodes by n. For a graph $G = (V, E)$, a subgraph $G' \subseteq G$, and nodes $u, v \in V(G')$, let $\pi(u, v, G')$ be the unique u-v shortest path in G' where shortest-path ties are decided arbitrarily in a consistent manner. Let $N(u, G)$ be the neighbors of node u in the graph G. When the graph G is clear from the context we may omit it and write $N(u)$. For a path $P = [u_1, \ldots, u_k]$ and an edge $e = (u_k, v)$, let $P \circ e$ denote the path obtained by concatenating e to P. Similarly, for two paths $P_1 = [u_1, \ldots, u_k], P_2 = [u_k, u_{k+1}, \ldots, u_\ell]$ denote the concatenated path $[u_1, \ldots, u_k, u_{k+1}, \ldots, u_\ell]$ by $P_1 \circ P_2$. Given a path $P = [u_1, \ldots, u_k]$ denote the sub-path from u_i to u_ℓ by $P[u_i, u_\ell]$. The term $\widetilde{O}(\cdot)$ hides $\mathrm{poly}(\log n)$ factors, and the term $\widehat{O}(\cdot)$ hides $2^{O(\sqrt{\log n})}$factors[10].

[9] To formalize this argument, we provide a formal definition for the cut value of a u-v walk collection.

[10] The latter factors arise by the (fault-free) distributed computation of cycle covers by [34].

▶ **Definition 10** (Neighborhood Covers, [2]). *The r-neighborhood cover of the graph G is a collection of vertex subsets, denoted as, clusters $\mathcal{N} = \{S_1, \ldots, S_\ell\}$ where $S_i \subseteq V$ such that: (i) every node v has a cluster that contains its entire r-radius neighborhood in G, (ii) the diameter of each $G[S_i]$ is $O(r \log^c n)$ for some constant c, and (iii) every node belongs to $\widetilde{O}(1)$ clusters in $\mathcal{N}$.*

We use the deterministic construction of neighborhood covers by Rohzon and Ghaffari [40] .

▶ **Theorem 11** (Corollary 3.5 [40]). *There is a deterministic distributed algorithm that for any radius $r \geq 1$, computes an r-neighborhood cover $\mathcal{N}$ within $\widetilde{O}(r)$ CONGEST rounds.*

Low-congestion cycle covers. The construction of FT cycle covers is based on the distributed construction of (c, d) cycle covers in the standard CONGEST model. In particular, we use the construction from [34, 33] that covers each edge $e = (u, v)$ by a cycle C_e such that $|C_e| = \widetilde{O}(\text{dist}_{G \setminus \{e\}}(u, v))$.

▶ **Fact 12** ([34, 33]). *There is a randomized algorithm* ComputeCycCov(G, D') *that for any n-node input graph $G = (V, E)$ and an input parameter D', computes, w.h.p., a cycle collection $\mathcal{C}$ with the following properties: (1) every edge $e \in E$ that lies on a cycle of length at most D' in G is covered by a cycle in $\mathcal{C}$ of length $\widehat{O}(D')$, and (2) each edge appears on $\widehat{O}(1)$ cycles. Algorithm* ComputeCycCov(G, D') *runs in $\widehat{O}(D')$ rounds. In the output format, each node knows the edges of the cycles that cover each of its incident edges.*

Note that Alg. ComputeCycCov does not require the graph G to be connected. This will be important in our context. This algorithm can also be made deterministic using the neighborhood covers of Theorem 11.

▶ **Observation 13.** *The algorithm* ComputeCycCov(G_i, D') *of Fact 12 can be made deterministic using the neighborhood covering algorithm of Theorem 11. Additionally, in the output format of the algorithm, each node u knows a $\widehat{O}(1)$-bit unique identifier for each of the cycles it belongs to, as well as a full description of the cycle, obtained from both directions.*

Covering families. Our distributed algorithms in the adversarial CONGEST model are based on communication over a collection of G-subgraphs that we denote as covering family. These families are used extensively in the context of fault-tolerant network design [1, 44, 11, 20, 30, 36, 9, 7, 24, 21].

▶ **Definition 14** ((L, t) Covering Families). *For a given graph G, a family of G-subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ is a (L, t)* covering family*, if for every edge $e = (u, v) \in E$ and every set $F \subseteq E$ where $|F| \leq t - 1$, such that*[11] $\text{dist}_{G \setminus F \cup \{e\}}(u, v) \leq L$*, there exists a subgraph G_i satisfying that (P1) $\text{dist}_{G_i \setminus (F \cup \{e\})}(u, v) \leq L$, and (P2) $(F \cup \{e\}) \cap G_i = \{e\}$.*

Throughout, we use the following observation from [21].

▶ **Observation 15** (Observation 8 from [21]). *Consider a D-diameter graph $G = (V, E)$ and assume that $u, v \in V$ are connected in $G \setminus F$ for some $F \subseteq G$. It then holds that* $\text{dist}_{G \setminus F}(u, v) \leq 2(|F| + 1) \cdot D + |F|$.

[11] We note that our definition slightly differs from that of [21], in the sense that for a pair $e = (u, v), F$, we require the graph G_i (see below) to contain a cycle of length at least L covering e, rather than an L-length u-v path.

We note that by Observation 15, if G is $(t+1)$ edge-connected, then a $(5tD, t)$ family satisfies (P1) and (P2) for any edge $(u,v) \in E$, and an edge set F of size at most $(t-1)$. For our purposes, it is required for the nodes to know the covering family in the following sense.

▶ **Definition 16** (Local Knowledge of a Subgraph Family). *A family of ordered subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ where each $G_i \subseteq G$, is* locally known *if given the identifier of an edge $e = (u,v)$ and an index i, u and v can locally determine if $e \in G_i$.*

▶ **Fact 17** ([24]). *Given a graph G and an integer parameter L, the following holds.*

1. *Given that all nodes share a seed $\mathcal{S}$ of $\widetilde{O}(1)$ random bits, there exists a 0-round randomized algorithm for locally computing a $(L,1)$-covering (ordered subgraph) family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ such that $\ell = \widetilde{O}(L)$, where the covering property holds w.h.p. Given the seed $\mathcal{S}$, index $i \in \{1, \ldots, \ell\}$ and an edge identifier (u,v), each node can locally determine if $(u,v) \in G_i$.*
2. *For every $t \geq 1$, there exists a 0-round deterministic algorithm for computing a (L,t) covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ such that $\ell = ((Lt \log n)^{t+1})$. This covering family is locally known.*

Broadcast against adversarial edges. Our algorithms for constructing FT-cycle covers make use of the broadcast algorithms of Hitron and Parter [21], which are resilient to adversarial edges. We will use the following facts.

▶ **Theorem 18** ([21] Broadcast against a Single Adversarial Edge). *Given a D–diameter, 3 edge-connected graph G and an unknown adversarial edge e', the following holds.*

1. *There exists a deterministic broadcast algorithm which delivers a message m_0 from a designated node s to all nodes in V within $\widetilde{O}(D^2)$ rounds. In addition, at the end of the algorithm, all nodes obtain a linear estimate for the* diameter *of the graph.*
2. *There exists a randomized broadcast algorithm which delivers a message m_0 from a designated node s to all nodes in V within $\widetilde{O}(D)$ rounds, provided that all nodes share $\widetilde{O}(1)$ random bits.*

In addition, the same bounds hold in the case where there are multiple sources holding the same broadcast message m_0.

▶ **Theorem 19** ([21] Broadcast against f Adversarial Edges). *There exists a deterministic broadcast algorithm against f adversarial edges, for D-diameter, $(2f+1)$ edge-connected graphs, with round complexity of $(tD \log n)^{O(t)}$. In addition, the same bound holds in the case where there are multiple sources holding the same broadcast message m_0.*

The broadcast algorithm of Theorem 18 also implies a leader election algorithm. For completeness the proof of the following claim is given in the full version.

▷ Claim 20. [Leader Election against an Adversarial Edge] Given a D–diameter, 3 edge-connected graph G and an adversarial edge e', assuming a linear upper bound $D' = cD$ on the diameter (for some constant $c \geq 1$), there exists a randomized algorithm AdvLeaderElection that w.h.p elects a single leader known to all nodes in the graph within $\widetilde{O}(D^2)$ rounds.

2 Compilers against a Single Adversarial Edge

We first describe the construction of $(1, e')$-FT cycle covers where e' is the adversarial edge. Then, we describe how to compile a single round using these cycles.

2.1 $(1, e')$-FT Cycle Covers

In this section, we prove Theorem 5. This section is devoted for showing the following key lemma that computes a $(1, e')$-FT cycle cover given a locally known covering family.

▶ **Lemma 21.** *Given is a* 3 *edge-connected graph G, with a fixed unknown adversarial edge e'. Let L be an integer satisfying that for every edge $e = (u, v)$ it holds that* $\mathrm{dist}_{G\setminus\{e,e'\}}(u, v) \leq L$. *Assuming that all nodes locally know a $(L, 1)$ covering family $\mathcal{G}$ of size ℓ, there exists a deterministic algorithm* ComputeOneFTCycCov *for computing a $(1, e')$ FT-cycle cover $\mathcal{C}$ with parameters $c = \widehat{O}(\ell), d = \widehat{O}(L)$ within $\widehat{O}(L^2 \cdot \ell)$ rounds.*

Since the computation of the $(L, 1)$ covering family is straightforward using known tools, we focus on proving Lemma 21. As a warm-up, we describe the construction assuming a reliable setting (with no adversarial edges). Then, we handle the real challenge of the $(1, e')$-FT cycle cover computation in the presence of an adversarial edge.

Warm-up: $(1, e')$-FT cycle covers in a reliable communication graph. The construction is based on applying the cycle cover algorithm of [34] on every subgraph G_i in the covering family $\mathcal{G}$. Specifically, given a locally known covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$, the algorithm proceeds in ℓ iterations. In each iteration i it applies the cycle cover algorithm ComputeCycCov(G_i, L) from Observation 13 on the graph G_i with a diameter estimation L, resulting in a cycle collection $\mathcal{C}_i$. The final cycle collection is given by $\mathcal{C} = \bigcup_{i=1}^{\ell} \mathcal{C}_i$, that is, the union of all cycles computed in the ℓ iterations. We next analyze the construction.

Correctness. The round complexity, the cycle length, and the edge congestion bounds follow immediately by the construction. It remains to show that the cycle collection $\mathcal{C}$ is indeed a $(1, e')$-FT cycle cover. To see this, consider a fixed pair of edges $e = (u, v), e'$. We will show that $\mathcal{C}$ contains a cycle $C_{e,e'}$ that contains e and does not contain e'. An iteration i is defined to be *good* for the edge pair e, e' if $e' \notin G_i$, $e \in G_i$ and $\mathrm{dist}_{G_i\setminus\{e\}}(u, v) \leq L$. Since, $\mathrm{dist}_{G\setminus\{e,e'\}}(u, v) \leq L$, due to the covering property of $\mathcal{G}$, there exists a good iteration i^* for every pair e, e'. We next show that e is successfully covered in iteration i^* by some cycle C_e. By the properties of Alg. ComputeCycCov, in iteration i^* the edge e is covered by a cycle C of length $\widehat{O}(L)$. In addition, as $e' \notin G_{i^*}$ this cycle does not contain e' as required.

Algorithm ComputeOneFTCycCov (Proof of Lemma 21). Given is a locally known covering family $\mathcal{G} = \{G_1, \ldots, G_\ell\}$. The algorithm works in ℓ iterations, where in iteration i it performs the computation over the subgraph G_i. Since $\mathcal{G}$ is locally known, every node knows its incident edges in G_i, and ignores messages from other edges in that iteration. Iteration i then consists of two steps. In the first step, the nodes apply Alg. ComputeCycCov(G_i, L) of Observation 13 over the graph G_i with diameter estimate L. This results in a cycle collection $\mathcal{C}'_i(u)$ for every node u. In the output format of Alg. ComputeCycCov, every cycle in $\mathcal{C}'_i(u)$ is presented by a tuple $(ID(C), C)$, where $ID(C)$ is the unique identifier of the cycle of size $\widehat{O}(1)$ bits, and C is the collection of the cycle edges[12]. Since e' might be in G_i, the cycles of $\mathcal{C}'_i(u)$ can be totally corrupted.

In the second step of iteration i, the nodes apply a verification procedure for their cycles in $\mathcal{C}'_i(u)$. Only verified cycles will then be added to the set of cycles $\mathcal{C}_i(u)$. In the analysis section, we show that for every reliable edge $e = (u, v) \neq e'$, there exists at least one cycle in $\mathcal{C}(u) = \bigcup_i^{\ell} \mathcal{C}_i(u)$ that covers e. The third step of the algorithm handles the remaining

[12] Recall that in Alg. ComputeCycCov(G_i, L), each node receives the cycle description C from both directions, i.e., from its two neighbors on C. In case a node u obtained distinct cycle descriptions from its two neighbors, it omits the cycle from its cycle collection $\mathcal{C}'_i(u)$.

adversarial edge, in case needed. We next elaborate on these steps in more details. We focus on iteration i where the nodes communicate over the graph $G_i \in \mathcal{G}$.

Step (1) of iteration i: Cycle cover computation. The (fault-free) cycle cover algorithm ComputeCycCov of Observation 13 is applied over the subgraph $G_i \in \mathcal{G}$, with parameter L. Since the graph G_i is locally known, each node can verify which of its incident edges lie on G_i and ignore the messages from the remaining edges. During the execution of Alg. ComputeCycCov(G_i, L), if a node u receives an illegal message, or different cycle descriptions with the same cycle ID, these messages are ignored, as well as future messages in that iteration. At the end of the execution of ComputeCycCov(G_i, L), each node u performs the following verification step on its output cycle set $\mathcal{C}'_i(u)$. The goal of this verification is to ensure each cycle in $\mathcal{C}'_i(u)$ corresponds to a legal cycle.

Step (2) of iteration i: Cycle verification. First, each node u performs a *local* inspection of its cycles in $\mathcal{C}'_i(u)$, and declares the iteration to be *faulty* if $\mathcal{C}'_i(u)$ contains at *least* one of the following:

1. A cycle of length $\widehat{\omega}(D)$;
2. An edge appearing in $\widehat{\omega}(1)$ cycles in $\mathcal{C}'_i(u)$;
3. A partial cycle (i.e., a walk rather than a cycle);
4. Inconsistency in a cycle description $(ID(C), C) \in \mathcal{C}'_i(u)$ as obtained through the two neighbors of u on C.

In the case where $\mathcal{C}'_i(u)$ is found to be faulty, u sets $\mathcal{C}_i(u) = \emptyset$, and will remain silent throughout this verification step. We will call such a node an *inactive* node. A node whose local inspection is successful is called *active*.

We now describe the global verification procedure for an active node u. The verification step is performed in *super-rounds* in the following manner. Each super-round consists of $c = \widehat{O}(1)$ rounds, which sets the upper bound on the number of cycles that an edge (u, v) participates in. A single super-round has the sufficient bandwidth to exchange a single message through an edge (u, v) for each of the cycles on which (u, v) lies. We then explicitly enforce that in each super-round, each node u sends over an edge (u, v) at most *one* message per cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ for which $(u, v) \in C$.

For a cycle $(ID(C), C) \in \mathcal{C}'_i(u)$, let v_C be the node with largest ID in the cycle description C obtained by u during Alg. ComputeCycCov(G_i, L). We note that the cycle description C is not necessarily correct, and in particular, it could be that $(ID(C), C) \notin \mathcal{C}'_i(v_C)$. For each cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ such that $u = v_C$ (the cycle's leader), it initiates the following verification steps.

(2.1) A leader v_C of a cycle $(ID(C), C) \in \mathcal{C}'_i(v_C)$ sends the verification message $ver(C) = (ID(C), ID(v_C), ver)$ along its two incident edges on this cycle (i.e., in the clock-wise and counter clockwise directions).

(2.2) The verification messages are then propagated over the cycles for $R = \widehat{O}(L)$ super-rounds, where $\widehat{O}(L)$ is the upper bound on the maximal cycle length. Upon receiving a verification message $ver(C) = (ID(C), ID(v_C), ver)$, an active node u sends $ver(C)$ to a neighbor $w \in N(u)$ if the following conditions hold: (1) $(ID(C), C_u) \in \mathcal{C}'_i(u)$ for some cycle C_u, (2) v_C is the node with the highest ID in C_u, (3) w is a neighbor of u in C_u, and (4) u received the message $ver(C)$ from its second neighbor on the cycle C_u.

(2.3) A leader v_C of a cycle C such that $(ID(C), C) \in \mathcal{C}_i(v_C)$, which did not receive the verification message $ver(C)$ from both its neighbors in C within R super-rounds, initiates a *cancellation message*, $cancel(C) = (ID(C), ID(v_C), cancel)$, and sends it to both its neighbors in C. This indicates to the nodes on this cycle that the cycle should be omitted from their cycle collection.

(2.4) The cancellation messages $cancel(C)$ are propagated over the cycle C for R super-rounds in the following manner. Let τ_i be the first super-round of Step (2.3). In this super-round, v_C may start propagating the message $cancel(C)$ (if the conditions of 2.3 hold). Note, however, that the cancellation messages might also originate at the adversarial edge e'. Step (2.4) handles the latter scenario by augmenting the cancellation messages $cancel(C)$ with distance information. For every node u let d^1_u, d^2_u be the u-v_C distance on C along the first (second) u-v_C path in C. Note that u can locally compute d^1_u, d^2_u using the cycle description of C. A vertex u upon receiving a $cancel(C)$ message from its neighbor v on C acts as follows. Let d^j_u be the length of the v_C-u path on C that passed through v. Then, if the message $cancel(C)$ is received at u from v in super-round $r_j = \tau_i + d^j_u$, u *accepts* the cancellation message and sends it to its other neighbor on C. All other cancellation messages received by u in later or prior super-rounds are dropped.

(2.5) A leader v_C of a cycle $(ID(C), C) \in \mathcal{C}_i(v_C)$ that received a cancellation message $cancel(C)$ that it did not initiate from only *one* direction (i.e., from exactly one of its neighbors on C), broadcasts a cancellation message $cancel(i)$, i.e., canceling iteration i, to all the nodes in the graph by using the broadcast algorithm of Theorem 18(1) over the graph G. Since there is only one broadcast message $cancel(i)$ to be sent on that iteration, possibly by many cycle leaders, this can be done in the same time as a single broadcast operation (i.e., within $\widetilde{O}(D^2)$ rounds).

(2.6) A node that *accepts* a cancellation message $cancel(i)$ via the broadcast algorithm, omits all cycles obtained in this iteration i.

At the end of the i'th iteration, every node u defines a verified cycle set $\mathcal{C}_i(u)$. A cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ is defined as *verified* by u if the following conditions hold (i) it received a verification message $ver(C)$ from both neighbors in C, (ii) any cancellation message $cancel(C)$ received by u has been dropped, and (iii) u accepted no cancellation message $cancel(i)$. Every verified cycle $(ID(C), C) \in \mathcal{C}'_i(u)$ is added to the set $\mathcal{C}_i(u)$. Thus, $\mathcal{C}_i(u)$ consists of all verified cycles passing through u computed in iteration i. This concludes the description of the i'th iteration. The output of each node u is $\mathcal{C}(u) = \bigcup_{i=1}^{\ell} \mathcal{C}_i(u)$.

Step (3): Covering the adversarial edge. For a node u, an incident edge (u, v) is considered by u to *covered* if there exists a tuple $(ID(C), C) \in \mathcal{C}(u)$ such that $(u, v) \in C$. The goal of the third and final step is to cover the remaining uncovered edges. Every node u and an uncovered edge (u, v), broadcasts the edge (u, v) using the deterministic broadcast algorithm of Theorem 18(1). In the analysis section, we show that if there is an uncovered edge then it must be the adversarial edge. The reason for broadcasting the edge (u, v) by its endpoints is to prevent the adversarial edge from initiating this step (despite the fact that all edges are covered). To cover (u, v), the endpoint with the larger identifier, say u, initiates a construction of a BFS tree T rooted at u in $G \setminus \{(u, v)\}$. Within $O(L)$ rounds, u and v learn the u-v tree path P. Then the cycle covering (u, v) is given by $C = (v, u) \circ P$. The cycle $(ID(C), C)$ is then[13] added to the cycle collection $\mathcal{C}(w)$ of every $w \in C$.

The correctness is deferred to the full version. We also show that the graph G is not required to be 3 edge-connected or with a bounded diameter. Our cycle cover algorithm has the

[13] The ID of the cycle C can obtained by appending the maximum ID vertex on C with a special tag indicating that the cycle is added in Step (3).

guarantee to cover every reliable edge that lies on a reliable short cycle in G. That is, we achieve the following.

▶ **Lemma 22.** *There exists a deterministic algorithm* DetComputeOneFTCycCov(G, L) *that given a graph G containing a single adversarial edge e' and a parameter L, returns a collection of cycles and paths $\mathcal{C}$ with the following property. Every reliable edge $(u, v) \neq e'$ for which* $\text{dist}_{G \setminus \{e',(u,v)\}}(u, v) \leq L$ *is covered by a reliable $\widehat{O}(L)$-length cycle $C \in \mathcal{C}$ such that $e' \notin C$.*

2.2 General Compilers Given $(1, e')$-FT Cycle Cover

We next show that our $(1, e')$-FT cycle cover with parameters (c, d) yields a general compiler that translates any r-round distributed algorithm $\mathcal{A}$ into an equivalent algorithm $\mathcal{A}'$ that works in the presence e'.

Compiler against a single adversarial edge (Proof of Theorem 6). The compiler works in a round-by-round fashion, where every round of $\mathcal{A}$ is implemented in $\mathcal{A}'$ using a phase of $O(c \cdot d^2)$ rounds. At the end of the i'th phase, all nodes will be able to recover the original messages sent to them in round i of algorithm $\mathcal{A}$.

Compilation of round i. Let $\mathcal{C}$ be the cycle collection of the $(1, e')$-FT cycle cover. Fix a round i of algorithm $\mathcal{A}$, and let $M_{u \to v}$ be the message sent from u to v for every pair of neighbors $e = (u, v) \in E$ during the i'th round. In the i'th phase of $\mathcal{A}'$, the node u sends v the message $M_{u \to v}$ through e and *all* u-v paths $\mathcal{P}_{u,v} = \{C \setminus \{e\} \mid C \in \mathcal{C}, e \in C\}$. When sending the messages, each node on a path $P \in \mathcal{P}_{u,v}$ sends at most one message targeted from u to v. If a node w is requires to send at least two different messages from u to v, it omits both messages and sends a null message Φ over the cycle.

At the end of phase i, each node v sets the message $\widetilde{M}_{u,v}$ as its estimate for the message $M_{u \to v}$ sent by u in round i of $\mathcal{A}$. The estimate $\widetilde{M}_{u,v}$ is defined by applying the following protocol. In the case that v receives an identical message $M \neq \Phi$ from u through all the paths in $\mathcal{P}_{u,v}$, then $\widetilde{M}_{u,v} \leftarrow M$. Otherwise, $\widetilde{M}_{u,v} \leftarrow M'$ where M' is the message v received over the direct edge (u, v).

Correctness. We show that at the end of phase i for every edge $(u, v) \in E$ it holds that $\widetilde{M}_{u,v} = M_{u \to v}$. Consider the following two cases.

Case $e = e'$ is the adversarial edge. Since all u-v paths in $\mathcal{P}_{u,v}$ are reliable, all messages received by v over these paths must be identical. Thus, all the messages that v receives through the paths are identical, and equal to $M_{u \to v}$. By the definition of the $(1, e')$-FT cycle cover, $\mathcal{P}_{u,v} \neq \emptyset$. Hence, v accepts the correct message.

Case $e \neq e'$ is reliable. The message that u receives from v through the direct edge e is $M' = M_{u \to v}$. By the definition of the $(1, e')$-FT cycle cover there exists a cycle $C \in \mathcal{C}$ covering e that does not contain e'. Hence, if all edges on C deliver the same message from u to v, it must be the message sent by u. Thus, if all messages v received through the paths $\mathcal{P}_{u,v}$ are identical and differ from Φ, they are equal to $M_{u \to v}$. Otherwise, v accepts the correct message $M' = M_{u \to v}$ delivered through the reliable edge (u, v).

Round complexity. Since each edge belongs to at most c cycles in the $(1, e')$-FT cycle cover $\mathcal{C}$, and as all cycles are of length at most d, the number of messages sent over an edge in a given phase is bounded by $c \cdot d$. Hence, each phase is implemented in $O(c \cdot d^2)$ rounds.

3 Compilers against Multiple Adversarial Edges

At the heart of the compilers lies the construction of (f, F^*)-FT cycle covers in the adversarial CONGEST model, that we describe in this section. The description of the compilers that exploit these cycles are deferred to the full version. Our main result is a deterministic construction of (f, F^*)-FT cycle covers* in the adversarial CONGEST model.

▶ **Lemma 23.** *Given is an $(2f+1)$ edge-connected graph G with a fixed subset of unknown adversarial edges F^* of size f. Assuming all nodes locally know an $(L = 7fD, 2f)$-covering family $\mathcal{G}$ of size ℓ, there exists an r-round algorithm* ComputeFTCycCov *for computing an (f, F^*)-FT cycle cover* with parameters $d = \widehat{O}(L)$, $c = \widehat{O}(\ell \cdot L^2)$, and $r = \widehat{O}(\ell \cdot (fD \log n)^{O(f)})$ in the adversarial* CONGEST *model.*

The proof of Theorem 8 follows by combining Lemma 23 and Fact 17. The algorithm ComputeFTCycCov uses an $(L, 1)$ covering family $\mathcal{G}$ with slightly different properties than those provided in Definition 14. Specifically, we use the following fact from [24].

▷ Claim 24 ([24]). Given a graph G and an integer parameter L, there exists a (deterministic) 0-round algorithm that allows all nodes to locally know a family of subgraphs $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ of size $\ell = \widetilde{O}(L^2)$ where for every edge $e = (u, v) \in G$ such that $\text{dist}_{G \setminus \{e\}}(u, v) \leq L$ there exists a subgraph G_i satisfying that (P1) $\text{dist}_{G_i \setminus (\{e\})}(u, v) \leq L$, and (P2') $e \notin G_i$.

Our Approach. Before presenting the algorithm, we provide the high level approach. Consider the following natural algorithm for computing an (f, F^*)-FT cycle cover. Let $\mathcal{G}$ be an $(L, 2f)$ covering family for $L = O(fD)$. By applying the (fault-free) algorithm ComputeCycCov from Fact 12 on each subgraph $G_i \in \mathcal{G}$, we have the guarantee that all the reliable edges $E \setminus F^*$ are covered successfully as required by Definition 4. The key challenge is in determining whether the adversarial edges are covered as well. In particular, it might be the case that an edge $e \in F^*$ mistakenly deduces that it is covered, leading eventually to an illegal compilation of the messages sent through this edge. Note that unlike $(1, e')$-FT cycle covers, here an edge is covered only if it is covered by cycles of sufficiently large "flow".

Our approach is based on reducing the problem of computing an (f, F^*)-FT cycle cover into the problem of computing $(1, e')$-FT cycle covers in *multiple* subgraphs for every $e' \in F^*$. Specifically, we define a covering family $\mathcal{G}$ with the following guarantee for each adversarial edges $e' \in F^*$: for every $F \subseteq G$, $|F| \leq f$, there exists a subgraph G_i containing a short cycle covering e' such that $G_i \cap (F^* \setminus \{e'\} \cup F) = \emptyset$. Since the covering guarantees for every $e' \in F^*$ are based on such "good" subgraphs G_i, it is safe to apply Alg. DetComputeOneFTCycCov (from Lemma 22) on these subgraphs (as they contain at most one adversarial edge). This approach also has a major caveat which has to do with the fact that the subgraph G_i is not necessarily two-edge connected, and might not even be connected. In the single edge case, Alg. DetComputeOneFTCycCov is indeed applied on the input graph which is 3 edge-connected.

Recall that Alg. DetComputeOneFTCycCov is based on performing a verification step of the cycles, at the end of which we have the guarantee that at most one edge, corresponding to the adversarial edge, might not be covered. The third step of that algorithm then covers this edge, in case needed, using its fundamental cycle in the BFS tree. When applying Alg. DetComputeOneFTCycCov on the subgraph G_i the situation is quite different. Since G_i is not necessarily connected, there might be potentially a large number of edges in G_i that are uncovered by cycles. Broadcasting the identities of these edges is too costly. For this reason, our algorithm applies the reduction in a more delicate manner.

Specifically, the algorithm applies Step (3) of Alg. DetComputeOneFTCycCov on the neighborhood cover of G_i (with radius of $O(fD)$). In addition, it attempts to cover with Alg. DetComputeOneFTCycCov only edges that lie on short cycles in G_i (i.e., to fix the issue that G_i is not two edge-connected). This guarantees that at most one edge activates the brute-force covering procedure in which the edge e get covered by a fundamental cycle of a BFS tree in $G \setminus \{e\}$. We next describe the algorithm in details.

Algorithm ComputeFTCycCov (Proof of Lemma 23). Let $\mathcal{G} = \{G_1, \ldots, G_\ell\}$ be a $(L, 2f)$-covering subgraph family that is locally known to all the nodes (from Definition 14). The algorithm iterates over the subgraphs in $\mathcal{G}$. In phase i, the algorithm considers the subgraph $G_i \in \mathcal{G}$ and applies two major steps. Let $E_i = \{e = (u, v) \in G_i \mid \text{dist}_{G_i \setminus \{e\}}(u, v) \leq L\}$ be the set of edges in G_i that are covered by a short cycle (of length at most $L + 1$) in G_i [14]. During the i'th phase, the goal is to cover the edges in E_i. The first step considers covering the reliable edges in $E_i \setminus F^*$, and the second step considers the adversarial edges $F^* \cap E_i$. Note that the endpoints of an edge e does not necessarily know if it belongs to E_i.

Step (1): Covering non-adversarial edges in G_i. The algorithm employs the deterministic $(1, e)$-FT cycle cover algorithm DetComputeOneFTCycCov(G_i, L') of Lemma 22 on the subgraph G_i with diameter estimate $L' = O(L \cdot \log^c n)$, where c is the constant of Definition 10 (in the analysis part, it will be made clear why L' is set in this manner). When executing Alg. ComputeOneFTCycCov(G_i, L'), Step (3) of that algorithm which covers the adversarial edge is omitted. In addition, in the verification step of Alg. ComputeOneFTCycCov(G_i, L') (Step 2.6), instead of using the broadcast algorithm of [21] against a single adversarial edge, we use the broadcast algorithm of [21] against f adversarial edges over the original graph G (see Theorem 19). If during the execution of Alg. ComputeOneFTCycCov(G_i, L'), a node u receives an illegal message or that it needs to send too many messages through its incident edges (i.e., that exceeds the allowed $\widehat{O}(L'^2)$ congestion bound of Alg. ComputeOneFTCycCov(G_i, L')), it cancels the i'th iteration in the following sense. It omits all its cycles computed in the i'th phase, and remains silent until the next phase.

For a node u, let $\mathcal{C}_i(u)$ be the cycle collection obtained by u during ComputeOneFTCycCov(G_i, L'). Every node u that did not cancel the i'th phase, adds the cycles in $\mathcal{C}_i(u)$ to its final cycle collection $\mathcal{C}(u)$. Recall that the output of Alg. ComputeOneFTCycCov is given by a collection of tuples $\mathcal{C}_i(u) = \{(ID(C), C)\}$. At the end of Step (1), a node u considers its incident edge (u, v) as *i-handled* if there exists $(ID(C), C) \in \mathcal{C}_i(u)$ such that $(u, v) \in C$.

Step (2): Covering the adversarial edges in G_i. The goal of this step is to cover the adversarial edges of $E_i \cap F^*$. At the beginning of the step, the nodes locally compute a family of subgraphs $\mathcal{G}_i = \{G_{i,1}, \ldots, G_{i,\ell_i}\}$ of size $\ell_i = \widetilde{O}(L^2)$ using Claim 24 with parameter L. The algorithm then proceeds in ℓ_i iterations, where in each iteration j the nodes perform the following sub-steps over the communication subgraph $G_{i,j} \in \mathcal{G}_i$.

(2.1) Compute an L neighborhood-cover $\mathcal{S}_{i,j} = \{S_{i,j,1}, \ldots, S_{i,j,k_{i,j}}\}$ by applying Theorem 11, and let $T_{i,j,q}$ be the spanning tree of each node subset $S_{i,j,q}$.

(2.2) An edge (u, v) is *short bridgeless* if (i) (u, v) is not i-handled in Step (1), and (ii) there exists a tree $T_{i,j,q}$ containing u and v. For every short bridgeless edge e, the algorithm adds a cycle $C_e = \pi(u, v) \circ e$ to the cycle collection, where $\pi(u, v)$ is a u-v path in $T_{i,j,q}$.

[14] Note that the set E_i is unknown to the nodes in G.

If during the execution of this step, a node u detects an incident edge with a congestion above the limit, it omits all the cycles obtained in this step from its cycle collection $\mathcal{C}(u)$ and proceeds to the next sub-iteration.

The correctness analysis is deferred to the full version.

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM (JACM)*, 42(4):844–856, 1995.

2 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast distributed network decompositions and covers. *J. Parallel Distributed Comput.*, 39(2):105–114, 1996.

3 Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

4 Piotr Berman, Krzysztof Diks, and Andrzej Pelc. Reliable broadcasting in logarithmic time with byzantine link failures. *Journal of Algorithms*, 22(2):199–211, 1997.

5 Piotr Berman and Juan A. Garay. Cloture votes: $n/4$-resilient distributed consensus in $t+1$ rounds. *Math. Syst. Theory*, 26(1):3–19, 1993.

6 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October – 1 November 1989*, pages 410–415. IEEE Computer Society, 1989.

7 Greg Bodwin, Michael Dinitz, and Caleb Robelle. Optimal vertex fault-tolerant spanners in polynomial time. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021*, pages 2924–2938, 2021.

8 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

9 Diptarka Chakraborty and Keerti Choudhary. New extremal bounds for reachability and strong-connectivity preservers under failures. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, pages 25:1–25:20, 2020.

10 Ran Cohen, Iftach Haitner, Nikolaos Makriyannis, Matan Orland, and Alex Samorodnitsky. On the round complexity of randomized byzantine agreement. In *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, pages 12:1–12:17, 2019.

11 Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 169–178. ACM, 2011.

12 Danny Dolev. The byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982.

13 Danny Dolev, Michael J. Fischer, Robert J. Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

14 Danny Dolev and Ezra N. Hoch. Constant-space localized byzantine consensus. In *Distributed Computing, 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings*, pages 167–181, 2008.

15 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

16 Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International Conference on Fundamentals of Computation Theory*, pages 127–140. Springer, 1983.

17 Mattias Fitzi and Ueli Maurer. From partial consistency to global broadcast. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 494–503, 2000.

18 Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM J. Comput.*, 27(1):247–290, 1998.

19 Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21–23, 2015*, pages 3–12, 2015.

20 Fabrizio Grandoni and Virginia Vassilevska Williams. Faster replacement paths and distance sensitivity oracles. *ACM Transactions on Algorithms (TALG)*, 16(1):1–25, 2019.

21 Yael Hitron and Merav Parter. Broadcast CONGEST algorithms against adversarial edges. In *Distributed Computing – 29th International Symposium, DISC 2021*, 2021.

22 Yael Hitron and Merav Parter. Broadcast CONGEST algorithms against adversarial edges. *CoRR*, abs/2004.06436, 2021. arXiv:2004.06436.

23 Damien Imbs and Michel Raynal. Simple and efficient reliable broadcast in the presence of byzantine processes. *arXiv preprint*, 2015. arXiv:1510.06882.

24 Karthik C. S. and Merav Parter. Deterministic replacement path covering. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021*, pages 704–723. SIAM, 2021.

25 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

26 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact byzantine consensus on undirected graphs under local broadcast model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 – August 2, 2019*, pages 327–336, 2019.

27 Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 275–282, 2004.

28 Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

29 Alexandre Maurer and Sébastien Tixeuil. On byzantine broadcast in loosely connected networks. In *Distributed Computing – 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, pages 253–266, 2012.

30 Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

31 Merav Parter and Eylon Yogev. Congested clique algorithms for graph spanners. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

32 Merav Parter and Eylon Yogev. Distributed algorithms made secure: A graph theoretic approach. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1693–1710, 2019.

33 Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1673–1692, 2019.

34 Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, pages 89:1–89:14, 2019.

35 Merav Parter and Eylon Yogev. Optimal short cycle decomposition in almost linear time. http://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/main-icalp-cycles-full.pdf, 2019.

36 Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 – August 2, 2019*, pages 107–116, 2019.

37 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

38 Andrzej Pelc and David Peleg. Broadcasting with locally bounded byzantine faults. *Inf. Process. Lett.*, 93(3):109–115, 2005.

39 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* SIAM, 2000.

40 Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 350–363. ACM, 2020.

41 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 304–313. Springer, 1989.

42 Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In *Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16-18, 1990, Proceedings*, pages 358–367, 1990.

43 Sam Toueg, Kenneth J Perry, and TK Srikanth. Fast distributed agreement. *SIAM Journal on Computing*, 16(3):445–457, 1987.

44 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):1–13, 2013.

Fast Arrays: Atomic Arrays with Constant Time Initialization

Siddhartha Jayanti
సిద్ధార్థ జయంతి
MIT CSAIL, Cambridge, MA, USA

Julian Shun
MIT CSAIL, Cambridge, MA, USA

Abstract

Some algorithms require a large array, but only operate on a small fraction of its indices. Examples include adjacency matrices for sparse graphs, hash tables, and van Emde Boas trees. For such algorithms, array initialization can be the most time-consuming operation. *Fast arrays* were invented to avoid this costly initialization. A *fast array* is a software implementation of an array, such that the entire array can be initialized in just constant time.

While algorithms for *sequential* fast arrays have been known for a long time, to the best of our knowledge, there are no previous algorithms for *concurrent* fast arrays. We present the first such algorithms in this paper. Our first algorithm is linearizable and wait-free, uses only linear space, and supports all operations – initialize, read, and write – in constant time. Our second algorithm enhances the first to additionally support all the read-modify-write operations available in hardware (such as compare-and-swap) in constant time.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases fast array, linearizable, wait-free, asynchronous, multiprocessor, constant time, space efficient, data structure

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.25

Funding *Siddhartha Jayanti*: NDSEG Fellowship from the US Department of Defense
Julian Shun: DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, a Google Faculty Research Award, a Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, and a JUMP Center co-sponsored by SRC and DARPA

1 Introduction

Arrays are the most fundamental data structure in computer science. Semantically, an *array* of length m is an object that supports the following interface:

- $\textsc{Initialize}(m, f)$: return an array $\mathcal{O}$ initialized to $\mathcal{O}[i] = f(i)$ for each $i \in [m]$.[1]
- $\mathcal{O}.\textsc{Read}(i)$: return $\mathcal{O}[i]$, if $i \in [m]$.
- $\mathcal{O}.\textsc{Write}(i, v)$: update $\mathcal{O}[i]$'s value to v, if $i \in [m]$.

Here, $\textsc{Initialize}()$ is the *constructor* method that creates the object, and $\textsc{Read}()$ and $\textsc{Write}()$ are the regular operations an array supports. Ordinarily, initialization is achieved by allocating an array of length m and looping through to initialize its entries, while reads and writes simply use the hardware load and store instructions. This standard implementation achieves a space complexity of $O(m)$, and time complexities of $O(m)$ for initialization and $O(1)$ for reads and writes. These time complexities are good for applications that eventually access most of the entries of the array. But, some applications – such as adjacency matrix

[1] For a positive integer m, we use the notation $[m] \triangleq \{0, 1, \ldots, m-1\}$.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 25; pp. 25:1–25:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

representations of sparse graphs, van Emde Boas trees, and certain hash tables – need to allocate a large array when only a small fraction of the array will eventually be accessed. The time complexities of such algorithms would improve drastically if we had *fast arrays*: arrays that support all three operations – READ(), WRITE(), and *even* INITIALIZE() – in just $O(1)$ worst-case time. Perhaps surprisingly, sequential fast array implementations have been known for decades, but, to the best of our knowledge, concurrent implementations do not exist. We design the first algorithms for concurrent fast arrays in this paper.

1.1 Sequential fast arrays: history and applications

Sequential algorithms for fast arrays date back to at least the 1970s [1,4,25]. In fact, the well known *folklore algorithm* for the problem (which we will revisit in Section 3) was alluded to in an exercise of the celebrated text by Aho, Hopcroft, and Ullman [1] and further described by Mehlhorn [25] and Bentley [4]; it achieves a deterministic worst-case time complexity of $O(1)$ for each of the three operations, while using only $3m + 1$ memory words. Fast arrays have been important to the efficiency of several algorithms. Notably:

- Fast arrays are used in implementations of *van Emde Boas trees* [6, 30] – associative arrays that store keys from a universe $\{1, 2, \ldots, u\}$ and support *insert*, *get*, and *delete* with a time complexity of just $O(\log \log u)$.
- Katoh et al. [21] note that Knuth employs fast arrays in the implementation of the hash table in his *Simpath* algorithm [22], which enumerates all simple paths between two vertices in a graph. Knuth uses the hash table to efficiently implement a certain data structure called ZDD (Zero-suppressed binary Decision Diagram) [27], which has many applications besides Simpath [18,22,28,29,34,35].
- When a sparse graph of n vertices and $m \ll n^2$ edges is represented using an adjacency matrix, mere initialization can take $\Theta(n^2)$ time with a traditional array. With a fast array however, the graph can be stored in just $O(m)$ time. Consequently, for a constant degree graph, storing the graph takes $O(n)$ time instead of $\Theta(n^2)$ time.

More generally, fast arrays can increase the asymptotic efficiency of algorithms that have higher space complexity than time complexity – just allocate all the space in one huge block in $O(1)$ time.

This range of applications has spurred a lot of research into fast arrays in recent years. A string of papers, starting with Navarro's work in 2012 and culminating in three back to back papers in 2017 by Hagerup and Kammer, Loong et al., and Katoh and Goto, have brought down the space complexity from $3m + 1$ to $m + 1$ using complex bit-packing and chaining techniques [13, 21, 23, 31, 32]. Fredriksson and Kilpeläinen recently studied the empirical running times of the more practical implementations of these sequential fast arrays [9].

1.2 Concurrent fast arrays

In contrast to sequential fast arrays, which have been well studied, there has been no prior work on concurrent fast arrays, to the best of our knowledge. In this paper, we propose and design algorithms for two variants of concurrent fast arrays:

- **Fast Array:** This is an implementation of an array which supports the standard operations – INITIALIZE(m, f), READ(i), and WRITE(i, v) – and satisfies two conditions. First, each operation is *linearizable*, i.e., it appears to take effect at some instant between its invocation and response [15]. Second, each operation is not only *wait-free* [14], but the process that executes the operation completes it in a constant number of its steps. The first condition ensures *atomicity*, and the second condition ensures *efficiency*.

- **Fast Generalized Array:** Besides *load* and *store*, modern architectures like x86 commonly support read-modify-write (RMW) primitives, such as Compare-and-Swap (CAS), Fetch-and-Add (FAA), and Fetch-and-Store (FAS) [17]. In fact, some of these primitives are indispensible for efficiency and even solvability of problems that arise in concurrent systems. For instance, implementing a wait-free queue is impossible using only loads and stores [14]. Mutex locks can be implemented using loads and stores, but constant RMR (remote memory reference) complexity implementations are impossible using only loads and stores [3, 7, 26].

  Since RMW primitives are supported by hardware and are essential for concurrent algorithms, it would be ideal if the components of the fast array can be manipulated using these primitives. For instance, when implementing a fast array $\mathcal{O}$ on a multiprocessor that supports CAS and FAS in hardware, a process π should not only be able to read $\mathcal{O}[i]$ and write to $\mathcal{O}[i]$, but should also be able to CAS $\mathcal{O}[i]$ and FAS $\mathcal{O}[i]$. We term such an array, which allows all hardware-supported operations to be applied to its components, a *generalized array*.

  Let $\mathcal{S}$ be the set of hardware-supported RMW primitives. A *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Write}(i, v)$ operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$.

1.3 Our contributions

In addition to defining the two types of concurrent fast arrays, our paper makes the following two principal contributions:

- We design an algorithm for the (standard) fast array. If p processes share a fast array of length m, our algorithm uses only $O(m + p)$ space. More generally, to instantiate and use k fast arrays (for any k) of lengths $m_1, \ldots, m_k$, our algorithm uses only $O(M + p)$ space, where $M = \sum_{j=1}^{k} m_j$.
- We enhance the above algorithm to design a fast generalized array. Its space complexity is the same as the previous algorithm's – $O(m + p)$ for a single array of length m, and $O(M + p)$ for multiple arrays of combined length M.

Both of the above algorithms require hardware support for read, write, and CAS.

2 Model

We work in the standard asynchronous shared memory multiprocessor model where p processes, numbered $0, \ldots, p - 1$ run concurrently but asynchronously, and each process is either performing an initializable array operation or is idle. The computation proceeds in steps, where an *adversarial scheduler* decides which process π takes the next step.

To provide synchronization, we assume the hardware compare-and-swap (CAS) synchronization primitive. The CAS operation on a memory word X with arguments *old* and *new* is called as follows: $\textsc{Cas}(X, \mathit{old}, \mathit{new})$. The operation is atomic and has the following behavior. If $X = \mathit{old}$, then X's value is updated to *new* and *true* is returned to indicate that the operation successfully changed the value; otherwise, if $X \neq \mathit{old}$, the value of X is not changed and *false* is returned. On modern x86 architectures, individual memory words are 64-bits, and so any hardware primitive can be applied on a standard 64-bit word. Usefully however, CAS can also be executed on 128-bit double-words, i.e., two adjacent words of memory [17]. We will make use of this feature. (Note that this 128-bit CAS operation is *not* DCAS – a primitive that does CAS on two non-adjacent memory locations, which is not supported in modern architectures.)

A data structure is *linearizable* if each operation can be assigned a unique *linearization point* between its invocation and return, and the return values of the operations are consistent with those of a sequential execution in which operations are executed in the order of linearization points [15]. Operations are *bounded wait-free* if there exists a bound b such that every invocation by a process π returns within b of π's own steps [14]. In the literature, data structures that are both linearizable and wait-free are called *atomic*.

We measure the efficiency of an algorithm by its worst-case work and space complexities. The *space complexity* of an algorithm is the total number of memory words that the algorithm uses. The *work complexity* of an operation by a process π, is the total number of steps executed by π between the invocation and return of that operation. Since work complexity is the natural generalization of time complexity to multiprocessors, it is often called *time complexity* in the literature; we adopt this convention and use the two terms interchangeably. Furthermore, as is standard [1, 4, 25, 30–33], we assume that it takes constant time to allocate an *uninitialized* array of any size n. We call an object implementation *fast* if every operation on that object takes only $O(1)$ time to execute in the worst-case. Our paper focuses on fast algorithms for arrays and generalized arrays.

3 Folklore Sequential Algorithm

Our concurrent algorithms are inspired by the folklore sequential algorithm, and so we present the pseudo-code for a folkore fast array object $\mathcal{O}$ in Algorithm 1, and describe it below.

Algorithm 1 The folklore algorithm for a sequential fast array.

```
procedure INITIALIZE($m, f$)
    $A \leftarrow$ new $array[m]$
    $B \leftarrow$ new $array[m]$
    $C \leftarrow$ new $array[m]$
    $f_{init} \leftarrow f$
    $X \leftarrow 0$

procedure READ($i$)
    if $0 \leq B[i] < X$ and $C[B[i]] = i$ then return $A[i]$ else return $f_{init}(i)$

procedure WRITE($i, v$)
    $A[i] \leftarrow v$
    if $B[i] < 0$ or $B[i] \geq X$ or $C[B[i]] \neq i$ then
        $C[X] \leftarrow i$
        $B[i] \leftarrow X$
        $X \leftarrow X + 1$
```

The method INITIALIZE(m, f) instantiates a new fast array $\mathcal{O}$ of length m. The implementation of the fast array uses three un-initialized arrays A, B, and C, each of length m, an integer X, and stores a pointer f_{init} to the function f. We call A the *principal array* and use $A[i]$ to hold the current value of the abstract element $\mathcal{O}[i]$ for each index i that has been *initialized*, i.e., written to at least once. The elements of $A[i]$ corresponding to uninitialized indices of $\mathcal{O}[i]$ hold their initial, arbitrary values. B, C, and X are used to keep track of which indices i have already been initialized (as we describe later).

Using the mechanism described above, implementing read and write becomes simple. READ(i) just returns $A[i]$ if i has been initialized, and $f(i)$ otherwise. Correspondingly, WRITE(i, v) simply writes $A[i] \leftarrow v$, and ensures that index i is marked as initialized.

The main difficulty of the algorithm lies in efficiently remembering which set of indices i have been initialized. We use the array C and the integer X to maintain this set as follows. If k indices have already been initialized, then we ensure that $X = k$ and that the sub-array $C[0, \ldots, X-1]$ holds the values of these initialized indices. Correspondingly, we spend constant time in the INITIALIZE() method to set $X \leftarrow 0$. Terminologically, we call C the *certification array*, call the elements of the array *certificates*, call the elements of the sub-array $C[0, \ldots, X-1]$ *valid*, and say that an index i is *certified* when it appears in the valid sub-array.

Maintaining A, C, and X is sufficient to get a *correct* implementation of an array, but not an *efficient* one. For efficiency, we need to distinguish between certified and un-certified indices in constant time. We use the array B for this purpose. In particular, whenever we certify a new index i in an element $C[j]$, we set $B[i] \leftarrow j$ to maintain the invariant that

$$\mathcal{I} \equiv \ \forall i \in [m], \ (0 \leq B[i] < X \text{ and } C[B[i]] = i \ \textit{ if and only if } \text{index } i \text{ is initialized}).$$

The check that $0 \leq B[i] < X$ ensures that $C[B[i]]$ is valid, while the check that $C[B[i]] = i$ ensures that this valid element of the certification array, indeed, certifies that index i is initialized.

4 Our Concurrent Fast-Array

The goal of this section is to design a linearizable wait-free fast-array that is both time and space efficient. We do so by building on the ideas of the folklore algorithm.

The folklore algorithm is built on two pillars: (1) the principal array A, which stores the values of initialized indices, and (2) the *certification mechanism* constituted by B, C, and X, which ensures that initialized indices can be identified in constant time by invariant $\mathcal{I}$. The principal array can easily be maintained in the concurrent setting, however the certification mechanism, which is the main workhorse of Algorithm 1, must be redesigned to cope with concurrency.

The difficulty of using the sequential certification mechanism with multiple processors stems from the contention on the variable X, and on the next available slot in the certification array, $C[X]$. In particular, if all p processors are concurrently performing different write operations on different un-initialized indices $i_0, \ldots, i_{p-1}$, then the old certification mechanism will direct all of them to the same location $C[X]$ in the certification array. Regardless of how the contention is resolved, only one index can fit into $C[X]$, meaning that $p-1$ processes will fail to certify their index by placing it in $C[X]$ and will thereby need to find an alternate location in the certification array. So, in the worst-case, only one process will finish its operation after all p processes do one unit of work each, meaning the algorithm will do $O(p)$ work per operation rather than $O(1)$.

We overcome this difficulty of adapting the certifying mechanism posed by contention on C and X by introducing four ideas that we detail below. Our first idea will eliminate the contention, thereby enabling constant time certifications and look-ups; however, it bloats the space complexity to $\Omega(m \cdot p)$. Our second and third ideas, in combination, eliminate this space overhead and bring the space complexity down to just $O(m+p)$, while ensuring that the time complexities of operations remain at just $O(1)$. Our fourth idea describes how to share resources in order to minimize the space complexity when multiple fast arrays are instantiated.

Individual certification arrays. Our first idea is to eliminate the universal C and X, and instead equip each process π with its own certification array c_π and a corresponding *control variable* $\mathrm{X}[\pi]$. Here, X is a one-dimensional array of length p that is indexed by process ids, and each c_π is an array of length m. Thus, process π certifies a new index i by performing three steps: (1) writing $c_\pi[\mathrm{X}[\pi]] \leftarrow i$, (2) setting $\mathrm{X}[\pi] \leftarrow \mathrm{X}[\pi] + 1$, and (3) writing $B[i] \leftarrow (\pi, \mathrm{X}[\pi] - 1)$. (While it will not yet be clear to the reader at this stage, the relative order of steps 2 and 3 is very important for the correctness of later ideas. We expand on this thought in the forthcoming Remark 1.) Unlike the act of certifying a new index which involves modifying certification arrays and control variables, the act of checking whether a given index i is certified only requires reading. We allow process π to freely read the arrays of other processes while checking if an index is certified. This idea of individual certification arrays by itself would lead to a concurrent fast-array algorithm; however, the space complexity of this algorithm is inherently super-linear. In particular, if all p processes concurrently write to a previously un-initialized location i, an adversarial scheduler can force each of them to certify that location in its own certification array; if this happens for each of the m indices, then each c_π must store m indices, leading to a total space complexity of $\Theta(mp)$.

Synchronization and walk-back. To reduce the space complexity induced by individual certification arrays, we must ensure that each index i is certified by at most one process, even if multiple processes perform concurrent writes to the same un-initialized index. To do this, we introduce two related ideas: *synchronization on B* and *walk-back.* That is, each processor π that wishes to certify an index i attempts to CAS (rather than write) the pair $(\pi, \mathrm{X}[\pi] - 1)$ – indicating the location in its certification array where i is certified – into $B[i]$. We orchestrate the update to $B[i]$ using CAS to ensure that at most one process gets a return value of *true*, indicating that *it* is the process that succeeded in certifying i. Each other process π, whose CAS to $B[i]$ fails, "walks back", i.e., it reclaims the location $c_\pi[\mathrm{X}[\pi] - 1]$ that it was going to use to certify i by decrementing $\mathrm{X}[\pi]$. Since each index is certified by at most one process, and each process has at most one certification location that it will walk back on at any given time, the total space used across all c_π arrays is $O(m + p)$.

Array doubling. Our synchronization and walk-back scheme guarantees that at most $O(m + p)$ space is *used* across all of the c_π arrays, but we do not *a priori* know how many locations each process π will use in its c_π array. To ensure that we allocate only as much space as we use, we employ the classic idea of *array-doubling* from sequential algorithms. We initially allocate constant sized c_π arrays. Each time c_π fills up, we replace it with a newly allocated array c'_π of length $2 \cdot c_\pi.len$ and copy over the old $c_\pi.len$ elements from c_π to c'_π. Note that we *do not* de-allocate the old array c_π when we switch to c'_π, since other processes could be accessing it; yet, since the sum of a geometric series is proportional to the largest term in the series, our total memory allocation for the c_π arrays is proportional to the amount of space we end up using. (Note that it is important to have a mechanism by which other processes can get access to the current array c_π, since the location of the array is changing whenever we double. We describe this detail when we discuss the pseudo-code in a later sub-section. We will also describe how to implement array doubling with worst-case, rather than amortized, constant time per operation in the same sub-section.)

Sharing the certification mechanism. Array doubling allows us to share a single certification mechanism across all fast-array objects that we initialize. In particular, if we have multiple fast-arrays $\mathcal{O}_1, \ldots, \mathcal{O}_k$, each $\mathcal{O}_j$ can simply maintain the two instance variables $\mathcal{O}_j.A$ and $\mathcal{O}_j.B$, and *share* the certification mechanism – $\forall \pi \in [p], (\mathrm{X}[\pi], c_\pi)$. All we need to do to enable this sharing is store a pointer $\&(\mathcal{O}_j.A[i])$ as the certificate that i is initialized in fast-array $\mathcal{O}_j$, rather than just store the index i. Since all fast array objects can share one certification mechanism, the space complexity of maintaining k fast-arrays $\mathcal{O}_1, \ldots, \mathcal{O}_k$ of sizes $m_1, \ldots, m_k$ is just $O(M + p)$, where $M \triangleq \sum_{j=1}^{k} m_j$.

▶ Remark 1 (the relative order of synchronizing and incrementing). When a process π, with next available certification location $x_\pi = \mathrm{X}[\pi]$, is certifying a new index i, we described that our algorithm follows three logical steps (not including potential walk-back): (1) *writing the certificate*: $\mathrm{c}_\pi[x_\pi] \leftarrow i$, (2) *incrementing* $\mathrm{X}[\pi]$: $\mathrm{X}[\pi] \leftarrow x_\pi + 1$, and (3) *synchronizing* on $B[i]$: attempting to CAS the value (π, x_π) into $B[i]$. At first glance, it may appear that step (3) can be executed before step (2). Indeed, if this were possible, then we could simplify our algorithm by avoiding walk-backs altogether, since a process π that fails the CAS could simply not increment $\mathrm{X}[\pi]$. However, as we mentioned earlier, the relative order of steps (2) and (3) is pivotal to correctness. We now explain why.

Consider a scenario where two processes π and τ are each performing the operation $\textsc{Write}(i, new)$ on a previously un-initialized location $\mathcal{O}[i]$ whose initial value is $\mathcal{O}[i] = f(i) = old$, with the order of steps (2) and (3) swapped. Then the following sequence of events can occur:

1. Both process π and τ write $A[i] \leftarrow new$, read the same old value $b \leftarrow B[i]$ (in anticipation of having to $\textsc{Cas}$ $B[i]$ in step (3)), and start the certification process.
2. π completes steps (1) and (3) and thereby successfully changes $B[i]$'s value to (π, x_π). However, i is still not certified since step (2) is yet to be done, and $\mathrm{X}[\pi] \not> x_\pi$.
3. τ completes steps (1) and (3), but because τ's $\textsc{Cas}$ in step (3) fails, it does not need to execute step (2), and it returns from its write operation.
4. Having finished its write operation, τ performs $\textsc{Read}(i)$, but sees that i is not yet certified (since π has not yet finished step (2)), and returns $f(i) = old$.

- This execution is not linearizable, since τ reads the value *old* in $\mathcal{O}[i]$ even after it finishes writing new.

Remark 1 establishes that a process π must increment $\mathrm{X}[\pi]$ before it synchronizes at $B[i]$ in the certification process. This means that we must indeed implement walk-back to achieve space efficiency. However, if walk-back is not implemented very carefully, it can lead to a nasty race condition. We describe this possible race condition, and how to overcome it, in the next subsection.

4.1 A tricky race condition that must be avoided

Until now, we have described the main ideas that propel our space-efficient fast-array implementation at a high-level. Of these ideas, individual certification arrays are straightforward to implement as suggested, and array-doubling requires only mild adaptation to work in the face of asynchronous concurrency. The idea of walk-back however can lead to a nasty race-condition if it is not implemented correctly. We describe this potential race, and how we overcome it below.

In order to understand the race condition, let us consider the following set-up. We have a freshly initialized fast-array $\mathcal{O}$ with just two locations $\mathcal{O}[0, 1]$, and initialization function $f(i) = old$. There are three processors π, τ, and ρ with: $\mathrm{X}[\pi] = 0$, $\mathrm{X}[\tau] = 0$, and $\mathrm{X}[\rho] = 0$. The processes will perform the following operations:

- π will perform $\mathcal{O}.\textsc{Write}(0, new)$ followed by $\mathcal{O}.\textsc{Write}(1, new)$
- τ will perform $\mathcal{O}.\textsc{Write}(0, new)$
- ρ will perform $\mathcal{O}.\textsc{Read}(0)$ followed by another $\mathcal{O}.\textsc{Read}(0)$

By design, both locations initially hold the value *old* and at some point in time will take on the value *new* and hold that value forever. However, the race condition will be that ρ's first read of index 0 will return *new*, while its second read will return *old*. The initial value of $B[0]$ – which can be arbitrary by design of the algorithm – is pivotal to achieving the race. In particular, we consider the initial value $B[0] = (\pi, 0)$. The initial values of $A[0, 1]$ and $B[1]$ can be arbitrary.

We describe the offending run below in a sequence of bullet points. When the relative order of certain operations do not matter, we may describe them all in the same bullet point.

- Recall that $B[0] = (\pi, 0)$, π is performing $\textsc{Write}(0, \mathit{new})$, and τ is performing $\textsc{Write}(0, \mathit{new})$.

1. π and τ both write $A[0] \leftarrow \mathit{new}$, and both read the initial value b of $B[0]$. (They will need this value b when they attempt to certify index 0 and do a Cas on $B[0]$ later.)
2. π and τ both conclude that index 0 is not certified yet, and thus wish to certify the location.

- Recall that $\mathrm{X}[\pi] = 0$.

3. π's next open certification location is 0, thus π writes $c_\pi[0] \leftarrow 0$, and increments $\mathrm{X}[\pi] \leftarrow 1$. π now stalls (before attempting to Cas its certification location $(\pi, 0)$ into $B[0]$).

- Notice that while π is not finished with its certification process, index 0 is already certified, since $B[0] = (\pi, 0)$ initially, and location $c_\pi[0]$ is valid and holds the value 0.

4. ρ does its first $\textsc{Read}(0)$ operation. That is, it reads $B[0] = (\pi, 0)$, checks that $c_\pi[0]$ is valid and that $c_\pi[0] = 0$ and thereby returns the value $A[0] = \mathit{new}$.
5. ρ starts its second $\textsc{Read}(0)$ operation. It starts its verification by reading $B[0] = (\pi, 0)$, and then stalls.
6. τ's next open certification location is 0, thus τ writes $c_\tau[0] \leftarrow 0$, increments $\mathrm{X}[\tau] \leftarrow 1$, and successfully Cases its certification location $(\tau, 0)$ into $B[0]$.

- Notice that index 0 is now certified by two certificates $c_\pi[0]$ and $c_\tau[0]$. $B[0] = (\tau, 0)$ identifies only the new certificate, but process ρ is about to check for the old certificate $c_\pi[0]$.

7. π attempts to finish its certification process by Casing $(\pi, 0)$ into $B[0]$. However, its Cas fails. Thus, π walks-back, and resets $\mathrm{X}[\pi] \leftarrow 0$. This completes π's write operation to index 0.
8. π does its entire $\textsc{Write}(1, \mathit{new})$ operation. That is, it writes $A[1] \leftarrow \mathit{new}$, writes $c_\pi[0] \leftarrow 1$, increments $\mathrm{X}[\pi]$ to 1, successfully Cases $(\pi, 0)$ into $B[1]$, and returns.
9. ρ now finishes its operation. Since it had previously read $B[0] = (\pi, 0)$ and $\mathrm{X}[\pi] = 1 > 0$, it checks $c_\pi[0]$, finds the value 1 there, concludes that index 0 is not certified, and thereby returns $f(0) = \mathit{old}$.

- This run cannot be linearized since $\mathcal{O}[0]$ was initially *old* and became *new*, but ρ reads its value to be *new* and subsequently re-reads the value to be *old*.

We observe that the cause of this race condition is the coincidental initial value of $B[0]$. In particular, $B[0]$'s (potentially arbitrary) initial value, happened to coincide with the exact location that process π would use to certify index 0 and later have to walk-back on. This coincidence, in turn, caused $B[0]$ to become certified during step (3), before π finished its certification operation by updating $B[0]$ with a Cas.

Tombstoning. Since we have only constant time to initialize $\mathcal{O}$, we *cannot* control the initial values of all the elements $B[i]$. However, we *can* control which location in the certification array π uses to certify $B[i]$. Therefore, we eliminate this nasty race condition as follows. If the initial value of $B[i]$ is (π, k), then we ensure that that particular process π does not attempt to use its certificate $c_\pi[k]$ to certify index i. If it so happens that $c_\pi[k]$ is the next available certificate for process π when process π is attempting to certify i, then we simply *tombstone* that location by writing a special null-value $c_\pi[k] \leftarrow \bot$, and use the next location $c_\pi[k+1]$ to certify i. This ensures correctness.

Furthermore, observe that for each location i, there is exactly one initial value $B[i] = (\pi, k)$ that references exactly one process π and one specific location k. Thus, at most m locations get tombstoned by our method across all processes, and the space complexity bound of $O(M + p)$ continues to hold true even in the worst-case. (We expect that tombstoning will occur only very rarely in practice.)

4.2 The pseudo-code and its description

Having described individual certification arrays, synchronization and walk-back, array-doubling, and tombstoning, we are ready to describe our fast-array algorithm. We present the pseudo-code as Algorithm 2, and describe it below.

Naming conventions. In order to distinguish between variables of different processes and operations that are performed by a particular process π, we use subscripts. For example, we denote a local variable x of process π by x_π, and denote a $\textsc{Read}()$ operation by process π as $\textsc{Read}_\pi()$. We use captial letters, such as A and X, to refer to arrays that all processes have the address of by default. Importantly, note that the pointer to the *current* certification arrays, c_π, follows the above convention, and by default only process π has access to the array. In order to allow other processes to access these arrays, our implementation stores a pair in the control variable $\mathrm{X}[\pi]$. So, initially $\mathrm{X}[\pi] = (0, c_\pi)$ (rather than $\mathrm{X}[\pi] = 0$).

In order to implement array doubling, we maintain a *next* certification array c'_π alongside the current array c_π. As such, initially $\mathrm{X}[\pi] = (0, c_\pi)$ and no process other than π has access to the array pointed to by c'_π. When c_π fills up entirely, we maintain the invariant that $c'_\pi[0, \ldots, c_\pi.len - 1] = c_\pi[0, \ldots, c_\pi.len - 1]$ and thus, we can simply replace $\mathrm{X}[\pi] = (x_\pi, c_\pi)$ by $\mathrm{X}[\pi] = (x_\pi, c'_\pi)$; we also rename c'_π as c_π because it has become the current array, and allocate a new (un-initialized) c'_π that is twice the length of the new current array. In order to ensure that $c'_\pi[0, \ldots, c_\pi.len - 1] = c_\pi[0, \ldots, c_\pi.len - 1]$ by the time c_π gets entirely full, we *transfer* two values from c_π to c'_π each time a new value is appended to c_π. We note that we *do not* de-allocate the old certification arrays because other processes could potentially be reading from them – this does not change the asymptotic space complexity. However, each process can store pointers to all of its arrays so that they can be de-allocated when the fast array is no longer needed. In our algorithm, we choose to start with a c_π of length 2. (Any other constant length would have sufficed.)

A line-by-line description of the code is as follows. $\mathcal{O}.\textsc{Initialize}_\pi(m_\pi, f_\pi)$ simply allocates the unintialized arrays A and B of length m_π (Lines 1 and 2), and stores the initialization function f_π as f_{init} for future use (Line 3). The elements of $A[i]$ will be used to hold the values of the abstract elements $\mathcal{O}[i]$. The elements of $B[i]$ will be used to hold process-index pairs (π, j); as a matter of convention, we call the first element in the pair $B[i].pid$ (process id) and the second element $B[i].loc$ (location).

$\textsc{Write}_\pi(i_\pi, v_\pi)$ first updates $A[i_\pi]$ to the new value v_π (Line 4). Since index i_π may not yet be certified, it calls $\textsc{Certify}_\pi(i_\pi)$ (Line 5). As we describe below, $\textsc{Certify}_\pi(i_\pi)$ only creates a new certificate for i_π if the index is not already certified.

Since $\textsc{Certify}_\pi(i_\pi)$ creates a new certificate (if necessary), and must perform the synchronization-CAS on $B[i_\pi]$ after creating a certificate, it must read the old value of $B[i_\pi]$. Thus, $\textsc{Certify}_\pi(i_\pi)$ starts by reading $b^{old}_\pi \leftarrow B[i]$ (Line 16). Certification should

Algorithm 2 Atomic fast array for p processes. Pseudo-code shown for an arbitrary process π.

```
Variables:
    For each process $\pi \in [p]$ the following variables are shared across *all* fast-arrays $\mathcal{O}$:
        • $c_\pi[0,1]$ is a pointer to an allocated un-initialized array of length 2.
        • $c'_\pi[0,\ldots,3]$ is a pointer to an allocated un-initialized array of length 4.
        • $k_\pi$ is a non-negative integer that is initialized to 0.
        • $\mathrm{X}[\pi]$ is a pair that is initialized to $(0, c_\pi)$.

    Each object $\mathcal{O}$ has three instance variables instantiated by $\textsc{Initialize}_\pi(m_\pi, f_\pi)$:
        • $A$ and $B$ are arrays of length $m_\pi$.
        • $f_{init}$ stores the initial value function.

    Each process $\pi \in [p]$ uses the following arbitrarily initialized temporary local variables:
        • $b_\pi, b_\pi^{old}$: hold (process id, array index) pairs.
        • $x_\pi$: holds an array index.
        • $c_\pi^{other}$: holds an array pointer.

procedure $\mathcal{O}.\textsc{Initialize}_\pi(m_\pi, f_\pi)$
    $A \leftarrow$ new array$[m_\pi]$
    $B \leftarrow$ new array$[m_\pi]$
    $f_{init} \leftarrow f_\pi$

procedure $\mathcal{O}.\textsc{Write}_\pi(i_\pi, v_\pi)$
    $A[i_\pi] \leftarrow v_\pi$
    $\textsc{Certify}_\pi(i_\pi)$

procedure $\mathcal{O}.\textsc{Read}_\pi(i_\pi)$
    if $\textsc{IsCertified}_\pi(i_\pi)$ then
        return $A[i_\pi]$
    else return $f_{init}(i_\pi)$

procedure $\mathcal{O}.\textsc{IsCertified}_\pi(i_\pi)$
    $b_\pi \leftarrow B[i_\pi]$
    if $0 \le b_\pi.pid < p$ then
        $(x_\pi, c_\pi^{other}) \leftarrow \mathrm{X}[b_\pi.pid]$
        if $0 \le b_\pi.loc < x_\pi$ and $c_\pi^{other}[b_\pi.loc] = \&A[i_\pi]$ then return *true*
    return *false*

procedure $\mathcal{O}.\textsc{Certify}_\pi(i_\pi)$
    $b_\pi^{old} \leftarrow B[i_\pi]$
    if $\textsc{IsCertified}_\pi(i_\pi)$ then return
    $(x_\pi, -) \leftarrow \mathrm{X}[\pi]$
    if $x_\pi \ge c_\pi.len$ then
        $c_\pi \leftarrow c'_\pi$
        $c'_\pi \leftarrow$ new array$[2 \cdot c_\pi.len]$
        $k_\pi \leftarrow 0$
    if $b_\pi^{old}.pid = \pi$ and $b_\pi^{old}.loc = x_\pi$ then
        $c_\pi[x_\pi] = \perp$
        $x_\pi \leftarrow x_\pi + 1$
    $c_\pi[x_\pi] \leftarrow \&A[i_\pi]$
    $\mathrm{X}[\pi] \leftarrow (x_\pi + 1, c_\pi)$
    while $k_\pi < 2x_\pi - c_\pi.len$ do
        $c'_\pi[k_\pi] \leftarrow c_\pi[k_\pi]$
        $k_\pi \leftarrow k_\pi + 1$
    if not $\textsc{Cas}(B[i_\pi], b_\pi^{old}, (\pi, x_\pi))$ then
        $\mathrm{X}[\pi] \leftarrow (x_\pi, c_\pi)$
```

only be done if i_π is not already certified, and so it calls IsCertified$_\pi(i_\pi)$ and returns immediately if location i_π is already certified (Line 17). Otherwise, it fetches the next location x_π that is free for creating a certificate (Line 18). Certification proceeds in five steps:

1. If the current certification array is full (check on Line 19), then the next array becomes the current one (Line 20), and a new (un-initialized) next array is allocated (Line 21). Finally, the local variable k_π – which is used to keep track of how many elements in the current array have already been transferred to the next array – is reset to 0 (Line 22).
2. At this point, we are sure that location $c_\pi[x_\pi]$ exists, and is free to certify a new index. Lines 23–25 tombstone this location and increment x_π if $B[i]$'s initial value happens to already hold the value (π, x_π). (This step eliminates the nasty race condition described earlier.)
3. Lines 26–27 certify location $A[i_\pi]$, by writing a pointer to $\&A[i_\pi]$ in $c_\pi[x_\pi]$ and updating the current array pointer and the length of the valid sub-array values in $\mathrm{X}[\pi]$.
4. Since a new location has been filled up in c_π, we must transfer values from c_π to c'_π. If there were no walking-back (described in the next step), then we would need to transfer exactly two values. However, because of potential walk-backs (in previous operations), it is possible that no values need to be transferred in this operation. Lines 28–30 orchestrate this transfer by maintaining k_π's progress relative to x_π.
5. Finally, π performs the synchronization step: it tries to finish its certification process with a Cas on Line 31. If the Cas fails, then it walks-back on Line 32.

That completes the description of Certify$_\pi(i_\pi)$.

Read$_\pi(i_\pi)$ simply checks whether element i_π is certified (Line 6), and returns $A[i_\pi]$ or $f_{init}(i_\pi)$ accordingly (Lines 7–8).

Like the read operation, IsCertified$_\pi(i_\pi)$ is also short. However, its code highlights an important point. The operation starts by reading $b_\pi \leftarrow B[i_\pi]$ which would hold the location of a certificate if i_π was initialized (Line 10). This is where it must be careful. The values in the pair $b_\pi = (b_\pi.pid, b_\pi.loc)$ are directly read from $B[i_\pi]$, and are thus potentially un-initialized ill-formed values. Thus, before the operation proceeds, it must check that $b_\pi.pid$ is indeed a real process id (Line 11). If so, it reads the control information in $\mathrm{X}[b_\pi.pid]$ to get a pointer to $c_\pi^{other} = c_{b_\pi.pid}$ and the length of its valid sub-array x_π. *After checking that* $b_\pi.loc$ is indeed in the valid portion of c_π^{other}, it verifies that the location $c_\pi^{other}[b_\pi.loc]$ certifies $A[i_\pi]$ (Line 13). The careful well-formedness checks are necessary to avoid accessing un-allocated portions of memory. Of course, the operation returns *true* only if all of the checks pass (Line 13); if *any* checks fail (well-formedness or otherwise), it simply returns *false* (Line 14).

▶ Remark 2 (old certification arrays). A process π that is performing IsCertified$_\pi(i_\pi)$ may hold a reference c_π^{other} that is no longer the current array of any process. That is, after π reads $(x_\pi, c_\pi^{other}) \leftarrow \mathrm{X}[b_\pi.pid]$, the process $b_\pi.pid$ may have certified more elements and updated its current certification array. However, our algorithm remains correct, since the old certification arrays are not de-allocated, and the value of $c_\pi^{other}[b_\pi.loc]$ is guaranteed to be equal to $c_{b_\pi.pid}[b_\pi.loc]$ – the corresponding value in the current array.

The preceding discussion is summarized in the theorem below.

▶ **Theorem 3.** *Algorithm 2 is a linearizable wait-free fast array implementation for p processes. That is, it supports* Initialize$_\pi(m_\pi, f_\pi)$*,* Read$_\pi(i_\pi)$*, and* Write$_\pi(i_\pi, v_\pi)$ *each with a time complexity of* $O(1)$*. The total space complexity of the algorithm for supporting k fast-arrays of sizes* $m_1, \ldots, m_k$ *is* $O(M + p)$*, where* $M = \sum_{j=1}^{k} m_j$*.*

5 A Concurrent Fast Generalized Array

In this section, we implement fast generalized arrays, which we motivated in Section 1.2.

Recall that, if $\mathcal{S}$ is the set of hardware-supported RMW primitives, then a *fast generalized array* is an implementation that not only supports $O(1)$-time linearizable $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Write}(i, v)$ operations, but also supports $O(1)$-time linearizable operations from the set $\mathcal{S}$. To this end, we consider the operation:

- $\mathcal{O}.\textsc{Apply}(i, op, args)$: perform operation *op* with arguments *args* on $\mathcal{O}[i]$, and return the response.

Here *op* can be any RMW operation – such as Write, CAS, FAA, or FAS – that is supported in hardware, and *args* are the arguments that the primitive requires. For example, if $\mathcal{O}[5] = 17$, then a call to $\mathcal{O}.\textsc{Apply}(5, \textsc{Cas}, (17, 35))$ changes the value of $\mathcal{O}[5]$ to 35 and returns *true*. We term an array that supports $\textsc{Initialize}(m, f)$, $\textsc{Read}(i)$, and $\textsc{Apply}(i, op, args)$, a *generalized array*, and an implementation that runs each operation in $O(1)$ time, a *fast generalized array*. Note that a $\textsc{Write}(i, v)$ can be executed as $\textsc{Apply}(i, \textsc{Write}, v)$. While $\textsc{Read}(i)$ can similarly be executed using $\textsc{Apply}(i, \textsc{Read})$, we design a simpler read method that circumvents the certification overhead for locations that are only read and never updated.

The goal of this section is to design a fast generalized array. We will achieve this goal by building on our ideas from Algorithm 2 for concurrent fast arrays. Therefore, we will continue to use the ideas of individual certification arrays, synchronization and walk-back, array doubling, sharing of certification arrays, and tombstoning. Even so, supporting arbitrary RMW operations poses yet new challenges. We first describe these challenges, and then explain how we overcome them.

Recall that at a high level, our fast array algorithm represents each abstract array element $\mathcal{O}[i]$ by the value of $A[i]$, along with the certification mechanism which keeps track of whether i has been initialized. Thus, $\textsc{Read}(i)$ simply returns $A[i]$ if i is initialized and $f(i)$ otherwise. Write operations, on the other hand, follow an *apply-then-certify* scheme. That is, $\textsc{Write}(i, v)$ blindly *applies* its operation by writing $A[i] \leftarrow v$, and subsequently *certifies* i if necessary.

A natural idea for implementing an RMW operation on $\mathcal{O}[i]$ would be to mimic the apply-then-certify scheme used by writes in Algorithm 2. For example, $\mathcal{O}.\textsc{Apply}(i, \textsc{Cas}, (old, new))$ would blindly apply $r \leftarrow \textsc{Cas}(A[i], old, new)$, and then certify i if necessary, and finally return r. Indeed, this idea would work if i were already initialized, since, in that case, $A[i]$ would hold the value of $\mathcal{O}[i]$. However, if i were not already initialized, then the abstract element $\mathcal{O}[i]$ has the value $f(i)$, while $A[i]$ has some arbitrary value. In particular, if $f(i) = old$, but $A[i] = \textit{some-other-value}$ (not equal to *old*), then $\mathcal{O}.\textsc{Apply}(i, \textsc{Cas}, (old, new))$ should change $\mathcal{O}[i]$'s value to *new* and return *true*, but the proposed scheme would keep the value the same (at $\mathcal{O}[i] = A[i] = \textit{some-other-value}$), certify index i, and return *false*. Thus, both the final value of $\mathcal{O}[i]$ and the return value of the operation would be incorrect.

From the example above, we see that RMW operations are difficult to apply before index i is initialized and certified, but easy to apply after the certification process. So, our idea is to reverse the scheme, rather than *apply-then-certify*, we will implement *certify-then-apply*. Since this new scheme will ensure that i is always certified first, the actual application of the RMW primitive can be realized as a hardware primitive applied directly to $A[i]$. Consequently, we can apply *any* primitive operation that hardware supports, not just the select few that we listed at the beginning of the section.

Certifying first poses a new challenge. When we applied writes to $A[i]$ before certifying, we were guaranteed that $A[i]$ would hold a valid (linearizable) value by the time i was certified. Since a reader will return $A[i]$ as the value of $\mathcal{O}[i]$ any time after i is certified, we still need to guarantee that $A[i] = \mathcal{O}[i]$ at the time of certification. This seems to be a difficult requirement with our current setup, since our previous certification process did not touch $A[i]$, but rather linearized at the time that a Cas was performed on $B[i]$. To overcome this challenge, we introduce the idea of *fusing* as described below.

Fusing. The values stored in each $A[i]$ correspond to the values stored in the corresponding abstract element $\mathcal{O}[i]$. So, it is important to allow these values to take up a full-pointer sized word, e.g., a 64-bit full-word in a modern 64-bit architecture. The pairs (π, j) that we are storing in $B[i]$ however, are just an internal representation used by our algorithm. Furthermore, it is entirely reasonable to assume that this pair can be stored in a single full-word. For example, allocating 14-bits for the process id π would allow for over 16,000 processors, and the remaining 50-bits would be enough to index an array with a thousand-trillion indices (i.e., an array taking up 8000 terabytes of memory). Therefore, we modify our representation by, intuitively, "absorbing the array B into A". Now, each element of our array $A[i]$ will hold a triple $(A[i].val, A[i].pid, A[i].loc)$, where the *value* $A[i].val$ is stored in the first word and the *process id* $A[i].pid$ and the *location* $A[i].loc$ are packed into the second word of a *double-width word*. Modern architectures, such as x86-64, allow us to perform double-width Cas operations on the full double-word $A[i]$, while also allowing all the standard single-width hardware primitives (Cas, Faa, Fas, Write, etc.) on the first word $A[i].val$. Using this feature of hardware, we can safely implement the "certify" portion of the certify-then-apply scheme. In particular, if process π reads $a_0 \leftarrow A[i]$ in its "un-initialized" state, and creates a certificate for it in $c_\pi[j]$, it can perform the certify step via: $\text{Cas}(A[i], a_0, (f(i), \pi, j))$.

5.1 The pseudo-code and its description

The pseudo-code for a process π's operations on our fast generalized array is presented as Algorithm 3. The algorithm is built on all of the ideas from the previous section – individual certification arrays, synchronization and walk-back, concurrent array-doubling, tombstoning, and certification mechanism sharing – along with the ideas introduced above – fusing, and the certify-then-apply scheme. We proceed to briefly describe the pseudo-code below.

The code of the three operations in the interface is simple to understand. $\mathcal{O}.\text{Initialize}_\pi(m_\pi, f_\pi)$ simply instantiates a single new un-initialized array A (Line 1), and stores the initialization function (Line 2). $\mathcal{O}.\text{Apply}_\pi(i_\pi, op_\pi, args_\pi)$ executes certify-then-apply by simply certifying at Line 3 and applying (and returning) at Line 4. $\mathcal{O}.\text{Read}_\pi(i_\pi)$ simply returns $A[i_\pi]$'s value field *val* if i_π is certified and $f_{init}(i_\pi)$ otherwise (Line 5).

Once again, the main workhorse of the algorithm is the certification mechanism. $\mathcal{O}.\text{Certify}_\pi(i_\pi)$ returns early if i_π is already certified (Lines 13–14). Otherwise, it loads the next available certification location x_π from $\text{X}[\pi]$ at Line 15, and follows the same logical steps as our earlier certification method: (1) update current arrays if necessary (Lines 16–19), (2) tombstone the location if the nasty race condition might arise (Lines 20–22), (3) create a certificate for $A[i_\pi]$ (Lines 23–24), (4) transfer values to the next array (Lines 25–27), and (5) synchronize, and walk-back if necessary (Lines 28–29). The most noteworthy difference from Algorithm 2 is Line 28, where we perform the double-width Cas operation to simultaneously update $A[i_\pi].val$ to $f_{init}(i_\pi)$ and $(A[i].pid, A[i_\pi].loc)$ to (π, x_π).

$\mathcal{O}.\text{IsCertified}_\pi(i_\pi)$ now reads a triple a_π (rather than the pair b_π) at Line 7, but performs the same logical function as in the standard fast array. It returns *true* only if $a_\pi.pid$ and $c_{a_\pi.pid}[a_\pi.loc]$ are valid, and if so certifies $A[i_\pi]$ (Lines 8–10). Otherwise, it returns *false* (Line 11).

The preceding discussion is summarized in the theorem below.

Algorithm 3 Atomic fast generalized array for p processes. Pseudo-code shown for an arbitrary process π.

```
Variables:
    For each process $\pi \in [p]$ the following variables are shared across all fast-arrays $\mathcal{O}$:
        • $c_\pi[0,1]$ is a pointer to an allocated un-initialized array of length 2.
        • $c'_\pi[0,\ldots,3]$ is a pointer to an allocated un-initialized array of length 4.
        • $k_\pi$ is a non-negative integer that is initialized to 0.
        • $X$ is an array, where each $\mathrm{X}[\pi]$ stores pair that is initialized to $(0, c_\pi)$.

    Each object $\mathcal{O}$ has two instance variables instantiated by INITIALIZE$_\pi(m_\pi, f_\pi)$:
        • $A$ is an array of double words.
        • $f_{init}$ stores the initial value function.

    Each process $\pi \in [p]$ uses the following arbitrarily initialized temporary local variables:
        • $a_\pi, a_\pi^{old}$: hold (value, process id, array index) triples.
        • $x_\pi$: holds an array index.
        • $c_\pi^{other}$: holds an array pointer.

procedure $\mathcal{O}$.INITIALIZE$_\pi(m_\pi, f_\pi)$
    $A \leftarrow$ new double-width-array$[m_\pi]$
    $f_{init} \leftarrow f_\pi$

procedure $\mathcal{O}$.APPLY$_\pi(i_\pi, op_\pi, args_\pi)$
    CERTIFY$_\pi(i_\pi)$
    return $op_\pi(A[i_\pi].val, args_\pi)$

procedure $\mathcal{O}$.READ$_\pi(i_\pi)$
    if ISCERTIFIED$_\pi(i_\pi)$ then return $A[i_\pi].val$ else return $f_{init}(i_\pi)$

procedure $\mathcal{O}$.ISCERTIFIED$_\pi(i_\pi)$
    $a_\pi \leftarrow A[i_\pi]$
    if $0 \le a_\pi.pid < p$ then
        $(x_\pi, c_\pi^{other}) \leftarrow \mathrm{X}[a_\pi.pid]$
        if $0 \le a_\pi.loc < x_\pi$ and $c_p^{other}[a_\pi.loc] = \&A[i_\pi]$ then return true
    return false

procedure $\mathcal{O}$.CERTIFY$_\pi(i_\pi)$
    $a_\pi^{old} \leftarrow A[i_\pi]$
    if ISCERTIFIED$_\pi(i_\pi)$ then return
    $(x_\pi, -) \leftarrow \mathrm{X}[\pi]$
    if $x_\pi \ge c_\pi.len$ then
        $c_\pi \leftarrow c'_\pi$
        $c'_\pi \leftarrow$ new array$[2 \cdot c_\pi.len]$
        $k_\pi \leftarrow 0$
    if $a_\pi^{old}.pid = p$ and $a_\pi^{old}.loc = x_\pi$ then
        $c_\pi[x_\pi] = \bot$
        $x_\pi \leftarrow x_\pi + 1$
    $c_\pi[x_\pi] \leftarrow \&A[i_\pi]$
    $\mathrm{X}[\pi] \leftarrow (x_\pi + 1, c_\pi)$
    while $k_\pi < 2x_\pi - c_\pi.len$ do
        $c'_\pi[k_\pi] \leftarrow c_\pi[k_\pi]$
        $k_\pi \leftarrow k_\pi + 1$
    if not CAS$(A[i_\pi], a_\pi^{old}, (f_{init}(i_\pi), \pi, x_\pi))$ then
        $\mathrm{X}[\pi] \leftarrow (x_\pi, c_\pi)$
```

▶ **Theorem 4.** *Algorithm 3 is a linearizable wait-free fast generalized array implementation for p processes. That is, for each process $\pi \in [p]$, it supports* $\textsc{Initialize}_\pi(m_\pi, f_\pi)$*,* $\textsc{Read}_\pi(i_\pi)$*, and* $\textsc{Apply}_\pi(i_\pi, op_\pi, args_\pi)$ *each with a time complexity of $O(1)$. The total space complexity of the algorithm for supporting k fast generalized arrays of sizes $m_1, \dots, m_k$ is $O(M + p)$, where $M = \sum_{j=1}^{k} m_j$, given that each memory word has at least $\lceil \log_2 M \rceil + \lceil \log_2 p \rceil$ bits.*

6 Discussion and Future Work

In this paper, we designed the first algorithms for concurrent fast arrays and fast generalized arrays. Just as sequential fast arrays have found several applications, we envisage future work that explores applications of these concurrent fast arrays. The following directions seem promising.

- The concurrent union-find data structure of Jayanti and Tarjan [19], which is used in the fastest parallel algorithms for computing connected components and spanning forests on CPUs and GPUs [8, 16], requires a generalized array of n nodes, with each node initially pointing to itself, i.e., $f(i) = i$. So, any concurrent union-find object on which only $o(n)$ operations are performed benefits from the use of our fast generalized array.
- A concurrent (standard) fast array is useful for implementing an adjacency matrix, E, of a mutable sparse graph. In particular, adding or removing an edge (i, j) is implemented by writing 1 or 0 (respectively) in $E[i, j]$, and querying an edge (i, j) is a simple read of $E[i, j]$. The real saving lies in storing the graph initially. To store a sparse graph of $m \ll n^2$ edges, we initialize the matrix E with all-zero entries in just $O(1)$ time, and then add the m edges, one at a time. Thus, the entire graph is stored in just $O(m)$ time, instead of the usual $\Theta(n^2)$ time.
- Kanellakis and Shvartsman introduced the *write-all* problem, a version of which is stated as follows: given an array A of length m such that each entry $A[i]$ has an arbitrary initial value, devise an algorithm for p asynchronous processes to initialize each entry $A[i]$ to 0, such that no process returns before the initialization is complete. This problem has attracted a lot of research [2, 5, 12, 20, 24], especially since a write-all solution is a critical subroutine in some implementations of concurrent hash tables [10, 11, 33].

  Although the two problems are different, fast arrays and write-all share the quest to achieve "fast initialization". The difference is that write-all insists on physically initializing each array element, whereas a fast array promises only to create the illusion of initializing each element. Thus, initialization takes only $O(1)$ time with fast arrays, while it takes at least linear time in any solution of write-all. Consequently, if an algorithm that uses a write-all solution can instead be satisfied with a fast array, then the algorithm's speed can potentially improve.
- Allocating a hash table of size n requires $\Theta(n)$ time using conventional arrays (because of initialization). As a result, it has been difficult to implement efficient re-sizable lock-free hash tables [10, 11, 33]. Using fast arrays however, a new table of any size can be allocated in just $O(1)$ time. Exploiting this feature, we are in the process of designing a re-sizeable wait-free hash table that guarantees $O(1)$ average time for *find* and *insert* operations.

We present some brief experiments to measure the empirical efficiency of our standard fast array algorithm in the appendix (Appendix A). We look forward to the further development and deployment of these ideas by algorithmists and practioners alike.

References

1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2 Richard J. Anderson and Heather Woll. Algorithms for the certified write-all problem. *SIAM J. Comput.*, 26(5):1277–1283, 1997.

3 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, page 447. ACM, 2008. doi:10.1145/1400751.1400843.

4 Jon Louis Bentley. *Programming pearls.* Addison-Wesley, 1986.

5 Jonathan F. Buss, Paris C. Kanellakis, Prabhakar L. Ragde, and Alex Allister Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.

6 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

7 Robert Cypher. The communication requirements of mutual exclusion. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.

8 Laxman Dhulipala, Changwan Hong, and Julian Shun. Connectit: A framework for static and incremental parallel graph connectivity algorithms. *Proc. VLDB Endow.*, 14(4):653–667, 2020.

9 Kimmo Fredriksson and Pekka Kilpeläinen. Practically efficient array initialization. *Softw. Pract. Exp.*, 46(4):435–467, 2016.

10 Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Almost wait-free resizable hashtable. In *International Parallel and Distributed Processing Symposium*, 2004.

11 Hui Gao, Jan Friso Groote, and Wim H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Comput.*, 18(1):21–42, 2005.

12 Jan Groote, Wim Hesselink, and Sjouke Mauw. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14:75–81, April 2001.

13 Torben Hagerup and Frank Kammer. On-the-fly array initialization in less space. In *International Symposium on Algorithms and Computation*, volume 92, pages 44:1–44:12, 2017.

14 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

15 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

16 Changwan Hong, Laxman Dhulipala, and Julian Shun. Exploring the design space of static and incremental graph connectivity algorithms on GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 55–69, 2020.

17 Intel. Intel 64 and IA-32 architectures software developer manuals, 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

18 Masakazu Ishihata, Shan Gao, and Shin-ichi Minato. Fast message passing algorithm using ZDD-based local structure compilation. In *Proceedings of the Workshop on Advanced Methodologies for Bayesian Networks*, volume 73, pages 117–128, 2017.

19 Siddhartha V. Jayanti and Robert E. Tarjan. A randomized concurrent algorithm for disjoint set union. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 75–82, 2016.

20 Paris C. Kanellakis and Alex A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, page 211–219, 1989.

21 Takashi Katoh and Keisuke Goto. In-place initializable arrays. *CoRR*, abs/1709.08900, 2017.

22 Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 12th edition, 2009.

23 Jacob Teo Por Loong, Jelani Nelson, and Huacheng Yu. Fillable arrays with constant time operations and a single bit of redundancy. *CoRR*, abs/1709.09574, 2017. arXiv:1709.09574.

24 C. Martel, R. Subramonian, and A. Part. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 590–599, 1990.

25 Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.

26 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

27 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the International Design Automation Conference*, page 272–277, 1993.

28 Shin-ichi Minato. Counting by ZDD. In *Encyclopedia of Algorithms*, pages 454–458. Springer Publishing Company, 2016.

29 Shin-ichi Minato. Power of enumeration—recent topics on BDD/ZDD-based techniques for discrete structure manipulation. *IEICE Trans. Inf. Syst.*, 100-D(8):1556–1562, 2017.

30 Dana Moshkovitz and Bruce Tidor. Lecture notes 15: van Emde Boas data structure. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec15.pdf.

31 Gonzalo Navarro. Constant-time array initialization in little space. In *Proceedings of the International Conference of the Chilean Computer Science Society (SCCC)*, 2012.

32 Gonzalo Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013.

33 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, 2006.

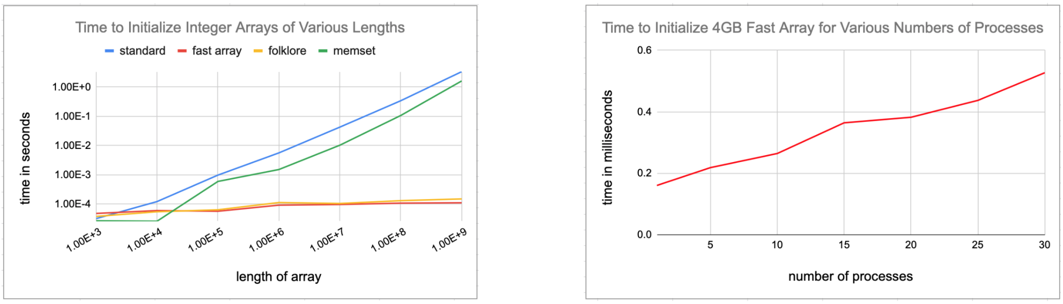
34 Hirofumi Suzuki and Shin-ichi Minato. Fast enumeration of all pareto-optimal solutions for 0-1 multi-objective knapsack problems using ZDDs. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 101-A(9):1375–1382, 2018.

35 Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.

A Experiments

We perform experiments using two 8-core Intel Xeon E5-2670 CPUs with two-way hyperthreading. The machine has 64GB of DRAM. Our machine ran 64-bit Ubuntu 12.04 with Linux kernel 3.13.0-143. All algorithms were coded in C++ and without any optimizations or specific algorithmic engineering to increase the speeds from the pseudo-code presented in the body of the paper. We used `std::threads` to implement our concurrent fast array, and we compiled our code with g++ version 4.8.4 with the `-std=c++11`, `-pthread`, and `-mcx16` options set. We experiment with four different algorithms:

1. *standard:* a classic array, where initialization is performed by a linear-time for-loop through the indices of the array.
2. *memset:* a classic array, where initialization is performed by the C++ `memset` primitive. The `memset` operation can only be used to initialize an array to all 0s. In particular, it cannot be used to initialize it to an arbitrary function f.
3. *folklore:* the sequential folklore fast array algorithm (i.e., Algorithm 1). This algorithm can only be used by a single process; it is not a concurrent algorithm, but it can serve as a baseline.
4. *fast array:* our concurrent fast array (i.e., Algorithm 2).



(a)

(b)

Figure 1 Figure 1a is a log-log plot charting the time to initialize arrays of various lengths for a single process. Figure 1b is a plot charting the times to initialize fast arrays of length one billion for various numbers of processes.

Our experiments focus on measuring the speeds of the three operations – Initialize(), Read(), and Write(). We present the results below.

1. To compare the speed of Initialize() across all four algorithms, we measure the time to initialize arrays to all zeroes. The array lengths we test are $m = 10^k$ for $k \in \{3, 4, 5, 6, 7, 8, 9\}$ with each entry being 4 bytes, i.e., arrays of size 4KB to 4GB. As predicted by the algorithmic analysis, the folklore and fast array algorithms take only constant time to initialize, while initializing by for-loop and `memset` take linear time.
   As shown in Figure 1a, initializing an array with `memset` is 1.2–4.7 times faster than initializing with a for-loop, however initializing a fast array of length one billion is more than 14,000 times faster than initializing with `memset`. As the length of the array gets smaller, the initialization time advantage of fast arrays reduces. Fast array initialization and `memset` initialization become equally fast at an array length between 10^4 and 10^5, and fast array initialization and for-loop initialization become equally fast at an array length between 10^3 and 10^4. We consistently observe that initializing a fast array takes only as much time as initializing a folklore array. As shown in Figure 1b, it takes only 3.3 times more time to initialize a 4GB fast array for 30 processes rather than one for a single process.
2. While fast arrays are much faster to initialize than standard arrays, read and write operations are slower on these arrays at all levels of concurrency. In order to measure exactly how much slower fast array reads are, we measure the cumulative time to perform one million reads using each type of array. Since the fast array Read() algorithm is different for indices that have been "initialized" – i.e., written to at least once after initialization – versus those that are "uninitialized", we measure the two types of reads separately (see Figure 2).
   The main takeaways of the experiment are as follows: (1) the two different types of reads – initialized and uninitialized – on fast arrays are of comparable speed; (2) fast array reads are 2.1–4.3 times slower than reads to standard arrays.
3. Algorithm 2 suggests that writes to "initialized" indices should be faster than those to "un-initialized" indices. This is also confirmed by our experiment shown in Figure 3. In particular, fast array writes to un-initialized indices are 6–21 times slower than standard writes, but writes to initialized indices are only 2.2–4.9 times slower. It is noteworthy here that the slower speed only occurs once per index, and all subsequent writes to that index happen at the faster speed.

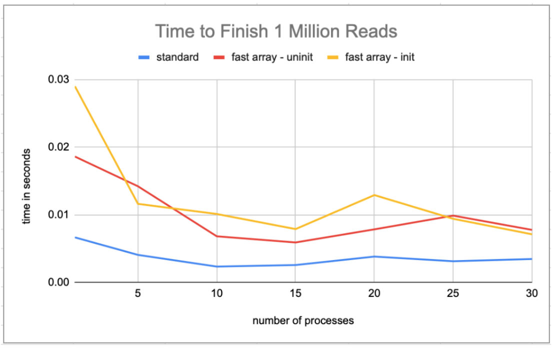


Figure 2 This plot compares the time to read from one million indices of a standard array versus a fast array for various numbers of concurrent processes. For the fast array, both reads from uninitialized and initialized indices are compared.

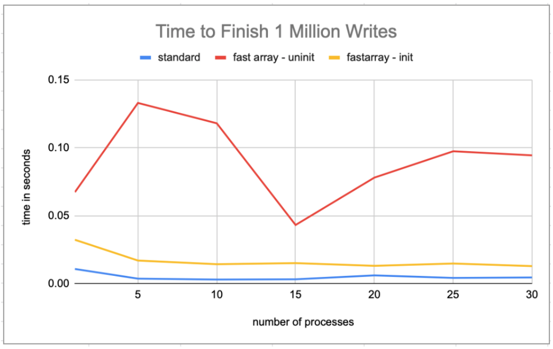


Figure 3 This plot compares the time to write to one million indices of a standard array versus a fast array for various numbers of concurrent processes. For the fast array, both writes to uninitialized and initialized indices are compared.

4. Writing to a new index in a fast array is 6–21× slower than writing to a new index in a naive array. So, fast arrays maintain their initial advantage over standard arrays as long as only 4–17% of the array indices are written to.

Byzantine Consensus with Local Multicast Channels

Muhammad Samir Khan ✉
Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA

Nitin H. Vaidya ✉
Department of Computer Science, Georgetown University, Washington, DC, USA

Abstract

Byzantine consensus is a classical problem in distributed computing. Each node in a synchronous system starts with a binary input. The goal is to reach agreement in the presence of Byzantine faulty nodes. We consider the setting where communication between nodes is modelled via an *undirected* communication graph. In the classical *point-to-point* communication model all messages sent on an edge are private between the two endpoints of the edge. This allows a faulty node to *equivocate*, i.e., lie differently to its different neighbors. Different models have been proposed in the literature that weaken equivocation. In the *local broadcast* model, every message transmitted by a node is received identically and correctly by all of its neighbors. In the *hypergraph* model, every message transmitted by a node on a hyperedge is received identically and correctly by all nodes on the hyperedge. Tight network conditions are known for each of the three cases.

We introduce a more general model that encompasses all three of these models. In the *local multicast* model, each node u has one or more local multicast channels. Each channel consists of multiple neighbors of u in the communication graph. When node u sends a message on a channel, it is received identically by all of its neighbors on the channel. For this model, we identify tight network conditions for consensus. We observe how the local multicast model reduces to each of the three models above under specific conditions. In each of the three cases, we relate our network condition to the corresponding known tight conditions. The local multicast model also encompasses other practical network models of interest that have not been explored previously, as elaborated in the paper.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine fault, distributed algorithm, consensus, broadcast, multicast

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.26

Related Version *Full Version*: `https://arxiv.org/abs/2109.01205` [10]

Funding This research is supported in part by the National Science Foundation award 1733872. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

1 Introduction

Byzantine consensus is a classical problem in distributed computing introduced by Lamport et al [12, 14]. There are n nodes in a synchronous system. Each node starts with a binary input. At most f of these nodes can be Byzantine faulty, i.e., exhibit arbitrary behavior. The goal of a consensus protocol is for the non-faulty nodes to reach agreement on a single output value in finite time. To exclude trivial protocols, we require that the output must be an input of some non-faulty node.

In this paper, we study consensus under the *local multicast* model. We formalize this model in Section 2. Intuitively, nodes are connected via an undirected graph G. A local multicast channel is defined by a sender and a set of receivers. Each node u may potentially serve as the sender on multiple local multicast channels. When node u sends a message on

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 26; pp. 26:1–26:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

one of its local multicast channels, it is received identically and correctly by all the receivers in the channel. This model generalizes the following models that have been considered before in the literature.

1. *Point-to-point communication model:* In the classical *point-to-point* communication model, each edge uv in the communication graph represents a private link between the nodes u and v. This model is well-studied [1, 4, 12, 13, 14]. It is well-known that $n \geq 3f+1$ and node connectivity at least $2f+1$ are both necessary and sufficient in this model.
2. *Local broadcast model:* Recently, we [8] studied consensus under the *local broadcast* model [2, 11], where a message sent by any node is received identically by all of its neighboring nodes in the communication graph. We obtained that minimum node degree at least $2f$ and node connectivity at least $\lfloor 3f/2 \rfloor + 1$ are both necessary and sufficient for Byzantine consensus under the local broadcast model [8].
3. *Hypergraph model:* A *hypergraph* is a generalization of graphs consisting of nodes and hyperedges. Unlike an edge in a graph, a hyperedge can connect any number of nodes. For a communication network modelled as a hypergraph, a message sent by a node u on a hyperedge e (that contains u) is received identically by all nodes in the hyperedge e. Communication networks modelled as hypergraphs have been studied in the literature [6, 7, 15]. Ravikant et al [15] gave tight conditions for Byzantine consensus on $(2,3)$-hypergraphs.[1] As discussed in Section 4, this result extends to general undirected hypergraphs as well.

The classical point-to-point communication model allows a faulty node to *equivocate* [3], i.e., send conflicting messages to its neighbors without this inconsistency being observed by the neighbors. For example, a faulty node z may tell its neighbor u that it has input 0, but tell another neighbor v that it has input 1. Since messages on each edge are private between the two endpoints, so node u does not overhear the message sent to node v and vice versa. The local broadcast model and the hypergraph model restrict a faulty node's ability to equivocate by detecting such attempts. In the local broadcast model, a faulty node's attempt to equivocate is detected by its neighboring nodes in the communication graph. In the hypergraph model, a faulty node's attempt to equivocate on a hyperedge is detected by the nodes in that hyperedge. In our local multicast model, a faulty node's attempt to equivocate on a single multicast channel is detected by the receivers in that channel.

In this work, we introduce the *local multicast* model, that unifies the models identified above, and make the following main contributions:

1. **Necessary and sufficient condition for local multicast model:** In Section 3, we present a network condition and show that it is both necessary and sufficient for Byzantine consensus under the local multicast model. The identified condition is inspired by the network conditions for directed graphs [9, 17], where node connectivity does not adequately capture the network requirements for consensus. We present a simple algorithm, inspired by [8, 9, 17].
2. **Reductions to the existing models:** The two extremes of the local multicast model are 1) each channel consists of exactly one receiver, and 2) each node has exactly one multicast channel. These correspond to the point-to-point communication model and the local broadcast model, respectively. In Section 4, we discuss how the network condition for the local multicast model reduces to the network requirements for the point-to-point

[1] i.e., each hyperedge consists of either 2 or 3 nodes.

model and the local broadcast model at the two extremes. On the other hand, if the multicast channels are induced from the hyperedges in a hypergraph, then this corresponds to the hypergraph model. In this case, the network condition reduces to the network requirements of the undirected hypergraph model given by Ravikant et al [15]. Moreover, our algorithm for the local multicast model works for all the three models identified here as well.

3. **Extensions to other models:** The local multicast model also captures some other models of practical interest (see Section 5). For instance, consider the scenario where nodes are connected via a wireless network. This can be modelled as local multicast over a graph G_1. Separately, the nodes are also connected via a bluetooth network, modelled using local multicast over a graph G_2 (with the same node set as G_1). Then the union of these networks $G_1 \cup G_2$ can be captured using the local multicast model as well. As another example, consider the scenario where nodes are connected via point-to-point channels. Additionally, nodes are also connected via a wireless network with local broadcast guarantees. As before, this can also be captured using the local multicast model. Our algorithm works for these cases as well.

In our recent work [10], we have generalized the results in this paper to the *directed* local multicast model. The directed local multicast model corresponds to directed hypergraphs where each directed hyperedge models a multicast channel with a single sender and a non-empty set of receivers. The tight condition obtained in [10] for the directed case is a natural extension of the tight condition obtained here for the undirected case. The results and proofs in [10] are more general and encompass the results in this paper.

2 System Model and Problem Formulation

We consider a synchronous system of n nodes. Nodes communicate using *local multicast channels*. Each node u has a set of multicast channels ζ_u. Each multicast channel $\chi_u \in \zeta_u$ is defined by the sender u and a non-empty set of receivers. For example, $\{v, w\} \in \zeta_u$ is a multicast channel of sender u with two receivers v and w. By convention used here, u is not included in the set of receivers. However, trivially, each node receives its own message transmissions as well. The communication between nodes is bidirectional so that if a node $v \in \chi_u$ for some channel $\chi_u \in \zeta_u$, then there exists a channel $\chi_v \in \zeta_v$ such that $u \in \chi_v$. A message m sent by a node u on a multicast channel χ_u is received identically and correctly by all nodes in χ_u. Moreover, each recepient $v \in \chi_u$ knows that m was sent by u on channel χ_u. We assume that each multicast channel is a FIFO communication channel.

The communication graph $G = (V(G), E(G))$ is an undirected graph where $V(G)$ is the set of n nodes and $uv \in E(G)$ is an edge of G if and only if there are channels χ_u and χ_v at nodes u and v, respectively, such that $u \in \chi_v$ and $v \in \chi_u$. Nodes u and v are *neighbors* in G. Observe that each multicast channel χ_u consists of a non-empty subset of the neighbors of u, such that each neighbor of u is in at least one channel in ζ_u.

- *Neighborhood:* For a set $S \subseteq V(G)$, a node $v \in V(G) - S$ is a neighbor of S if it is a neighbor of some node $u \in S$. More generally, for two disjoint sets $A, B \subseteq V(G)$, $\Gamma_G(A, B)$ defined below is the set of neighbors of B in A.

$$\Gamma_G(A, B) := \big\{u \in A \mid \exists v \in B : uv \in E(G)\big\} .$$

- *Adjacent:* For two disjoint sets $A, B \subseteq V(G)$, we use $A \rightarrow_G B$ (read as A is "adjacent" to B in G) to denote that either

(i) $B = \emptyset$, or

(ii) nodes in B have at least $f+1$ neighbors in A in the graph G, i.e.,

$$\left|\Gamma_G(A,B)\right| \geq f+1.$$

A *Byzantine faulty* node may exhibit arbitrary behavior. In *Byzantine consensus problem* each node starts with a binary input and must output a binary value satisfying the following constraints, in the presence of up to f Byzantine faulty nodes.

1. **Agreement:** All non-faulty nodes must output the same value.
2. **Validity:** If a non-faulty node outputs $b \in \{0,1\}$, then at least one non-faulty node must have input b.
3. **Termination:** All non-faulty nodes must decide in finite time.

It is easy to show that $f < n$ is necessary for Byzantine consensus. So we assume $f < n$ throughout the paper.

Node split

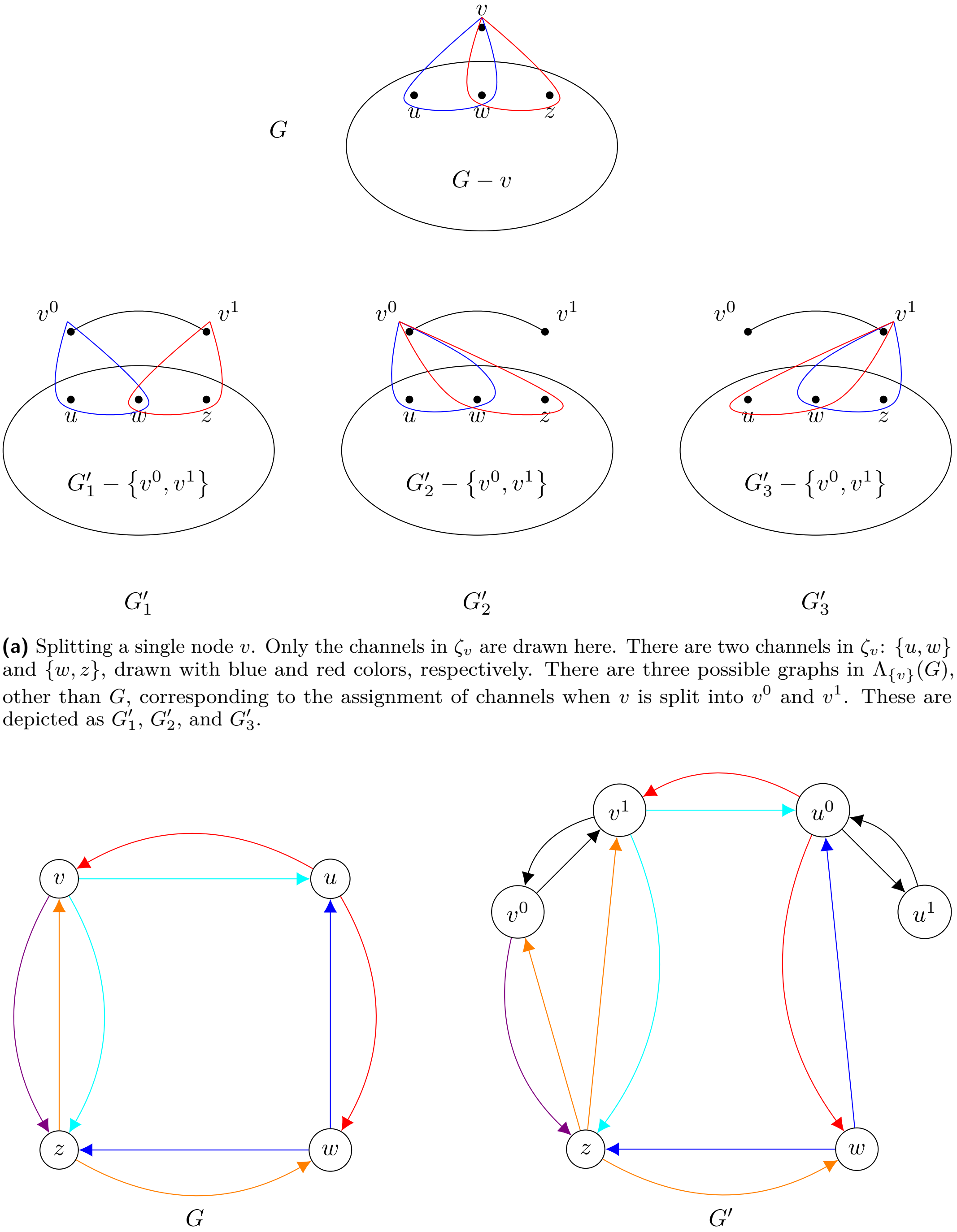
We now introduce the notion of a *node split* that is used to specify the necessary and sufficient condition under the local multicast model. As seen later, we will use the notion of node split to simulate possible equivocation by a faulty node. Intuitively, by splitting a node v, we are creating two copies of v and dividing up the channels amongst the two copies. Figure 1 shows two examples of node split. Formally, splitting a node v in G creates a new graph G' as follows.

- The node v is replaced by two nodes v^0 and v^1.
- We add an edge v^0v^1 to $E(G')$.
- We add a multicast channel $\left\{v^1\right\}$ to v^0 and a multicast channel $\left\{v^0\right\}$ to v^1.
- For every multicast channel χ_v of node v in G, choose exactly one of v^0 and v^1 as node v'. Create a multicast channel $\chi'_{v'}$ of v' with $\chi'_{v'} = \left\{u \mid u \in \chi_v\right\}$, i.e., each neighbor of v in χ_v is assigned to $\chi'_{v'}$.
- The above step adds edges to $E(G')$, of the form uv' such that $v' \in \left\{v^0, v^1\right\}$, but v' is not assigned to any multicast channel at node u. We specify these assignments as follows. Consider an edge uv', for $v' \in \left\{v^0, v^1\right\}$, in G'. For each multicast channel χ_u of node u in G, such that $v \in \chi_u$, add v' to the corresponding multicast channel χ'_u in G'. Now each neighbor w of u in G' is part of at least one multicast channel at node u.

Observe that for every node $u \in V(G')$, each of its multicast channels in G' corresponds to a single multicast channel in G, except for the two channels $\left\{v^1\right\}$ and $\left\{v^0\right\}$ at nodes v^0 and v^1, respectively (where node v was split). Similarly, for every node $u \in V(G)$, each of its multicast channels in G corresponds to a single multicast channel in G'.

To split two nodes u and v in G, we first split u to obtain G'. We then split v to obtain G'' from G'. The order of splits does not matter. This process naturally extends to splitting multiple nodes as well. For a set $F \subseteq V(G)$, let $\Lambda_F(G)$ be the set of all graphs that can be obtained from G by splitting some subset of nodes in the set F. For a graph $G' \in \Lambda_F(G)$, we use F' to denote the set of nodes in G' that correspond to nodes in F in G, i.e.,

$$F' := \left(V(G') \cap F\right) \cup \left(V(G') - V(G)\right).$$



(a) Splitting a single node v. Only the channels in ζ_v are drawn here. There are two channels in ζ_v: $\{u, w\}$ and $\{w, z\}$, drawn with blue and red colors, respectively. There are three possible graphs in $\Lambda_{\{v\}}(G)$, other than G, corresponding to the assignment of channels when v is split into v^0 and v^1. These are depicted as G'_1, G'_2, and G'_3.

(b) Splitting two nodes u, v in a 4-node graph G. Directed edges of the same color, pointing out from the same sender node, represent a single channel. G' is obtained by splitting nodes u and v into u^0, u^1 and v^0, v^1, respectively. The cyan channel is assigned to v^1, the violet channel is assigned to v^0, and the red channel is assigned to u^0.

Figure 1 Examples of the node split operation.

Note that there are two choices in the node split operation above which give rise to all the graphs in $\Lambda_F(G)$:

1. choice of which nodes in F to split, and
2. assignment of multicast channels for each split node.

As needed, we will occasionally clarify these choices to specify how a graph $G' \in \Lambda_F(G)$ was constructed by splitting some nodes in F.

3 Main Result

The main result of this paper is a tight characterization of network requirements for Byzantine consensus under the local multicast model. Consider a graph $G' \in \Lambda_F(G)$ obtained from G by splitting some nodes in a set F. Recall that we use F' to denote the set of nodes in G' that correspond to nodes in F in G.

▶ **Theorem 1.** *Under the local multicast model, Byzantine consensus tolerating at most f faulty nodes is achievable on graph G if and only if for every $F \subseteq V(G)$ of size at most f, every $G' \in \Lambda_F(G)$ satisfies the following: for every partition*[2] *(L, C, R) of $V(G')$, either*

1. *$L \cup C \rightarrow_{G'} R - F'$, or*
2. *$R \cup C \rightarrow_{G'} L - F'$.*

While we allow a partition to have empty parts, the interesting partitions are those where both L and R are non-empty, but C can be possibly empty. In Section 4, we show that when the local multicast model corresponds to the point-to-point, local broadcast, or hypergraph model, the above condition reduces to the corresponding known tight network conditions in each of the three cases.

We prove the necessity of Theorem 1 in Section 6. In Section 7, we give an algorithm to constructively show the sufficiency. The above condition is similar to the network condition for directed graphs in the point-to-point communication model [16, 17] and in the local broadcast model [9]. Note that [9, 16, 17] deal with consensus on arbitrary *directed* graphs, where connectivity constraints do not adequately capture the tight network requirements. In this paper, we are interested in undirected graphs. However, since the local multicast model is quite general and captures various models with different connectivity requirements, it is plausible that no concise network connectivity property will be able to properly characterize the tight condition.

For convenience, we give a name to the condition in Theorem 1.

▶ **Definition 2.** *A graph G satisfies* condition LCR with parameter F *if for every $G' \in \Lambda_F(G)$ and every partition (L, C, R) of $V(G')$, we have that either*

1. *$L \cup C \rightarrow_{G'} R - F'$, or*
2. *$R \cup C \rightarrow_{G'} L - F'$.*

We say that G satisfies condition LCR, *if G satisfies condition LCR with parameter F for every set $F \subseteq V(G)$ of cardinality at most f.*

[2] with a slight abuse of terminology, we allow a partition of a set to have empty parts.

4 Reductions to Other Models

In this section, we discuss how condition LCR relates to the tight conditions for the classical point-to-point communication model, the local broadcast model, and the hypergraph model.

Point-to-piont Channels

The classical point-to-point communication model corresponds to the case where each multicast channel in the graph consists of a single receiver node, so that the communication on an edge uv is private between the two nodes u and v. Under the point-to-point communication model, it is well known that $n \geq 3f + 1$ [5, 12, 14] and node connectivity at least $2f + 1$ [4, 5] are both necessary and sufficient for consensus on arbitrary undirected graphs.

When G has only point-to-point channels, i.e., each multicast channel consists of a single receiver node, then G satisfies condition LCR if and only if $n \geq 3f + 1$ and G has node connectivity $\geq 2f + 1$. We prove this formally in [10]. Therefore, the two models are equivalent when only point-to-point channels are present.

Local Broadcast

The local broadcast model corresponds to the other extreme where each node in the graph has exactly one multicast channel, so that the messages transmitted by a node u are received identically and correctly by all neighbors of u. Under the local broadcast model, we [8] showed that node degree at least $2f$ and connectivity at least $\lfloor 3f/2 \rfloor + 1$ are both necessary and sufficient for consensus on arbitrary undirected graphs.

When G has only local broadcast channels, i.e., each node has a single multicast channel, then G satisfies condition LCR if and only if G has minimum node degree $\geq 2f$ and node connectivity $\geq \lfloor 3f/2 \rfloor + 1$. We prove this formally in [10]. Therefore, the two models are equivalent when only local broadcast channels are present.

Hypergraphs

The last model we consider in this section is the hypergraph model. In a hypergraph $H = (V(H), E(H))$, each hyperedge $e \in E(H)$ is a subset of nodes $e \subseteq V(H)$. A hyperedge $e \in E(H)$ is called an $|e|$-hyperedge. Each hyperedge is effectively a multicast channel, i.e., a message sent by a node u on an edge $e \supseteq \{u\}$ is received identically and correctly by all nodes $v \in e$. However, any node on a hyperedge can act as a sender for this channel. In our local multicast model with communication graph G, this corresponds to the case where, for every pair of nodes u, v and multicast channel χ_u of u such that $v \in \chi_u$, there exists a channel χ_v of v such that $\chi_v = \big(\chi_u \cup \{u\}\big) - v$.

Ravikant et al. [15] obtained tight conditions for the hypergraph model. We observe that while the conditions were presented as a tight characterization for $(2, 3)$-hypergraphs[3] in [15], they also hold for general hypergraphs. In our local multicast model, when the communication graph G and its local multicast channels correspond to an undirected hypergraph, then condition LCR reduces to the tight conditions for hypergraphs given in [15]. The formal proof is given in [10].

[3] H is a $(2, 3)$-hypergraph if each hyperedge is either a 2-hyperedge or a 3-hyperedge.

5 Application to New Models

As mentioned in Section 1, the local multicast model also encompasses some other network models of practical interest that, to the best of our knowledge, have not been considered before in the literature. Suppose the n nodes are connected via a local multicast network with graph G_1. For example, network connectivity in G_1 can be via point-to-point links or via wireless channels modelled as local broadcast. Additionally, the n nodes are connected via another local multicast network with graph G_2. For example, G_2 may correspond to a wireless network with different frequencies and/or technologies. The complete system, where nodes can communicate on channels in G_1 as well as on channels in G_2, can also be characterized by the local multicast model. We omit details for brevity, but this corresponds to the natural union of G_1 and G_2, with each node now having access to its multicast channels in G_1 as well as its multicast channels in G_2.

6 Necessity of Condition in Theorem 1

Intuitively, consider a set $F \subseteq V(G)$ of size at most f, such that the graph G violates condition LCR with parameter F. With F as a candidate faulty set, the splitting of nodes in F captures possible equivocation by nodes in F: a faulty node can behave as if it has input 0 on some of its multicast channels and behave as if it has input 1 on the other multicast channels. Let $G' \in \Lambda_F(G)$ be a graph obtained by splitting nodes in F. We use F' to denote the nodes in G' that correspond to nodes in F in G. Suppose (L, C, R) is a partition of G'. Now consider the execution where non-faulty nodes in L have input 0. Since $R \cup C \not\rightarrow_{G'} L - F'$, nodes in $L - F'$ can not distinguish between F and its neighbors in $R \cup C$, i.e., $\Gamma_{G'}(R \cup C, L - F')$ as the set of faulty nodes. So non-faulty nodes in L are stuck with outputting 0 in this case. Similarly if non-faulty nodes in R have input 1, then they have no choice but to output 1, creating the desired contradiction.

A formal necessity proof is given in [10] for the *directed* local multicast model, which generalizes the *undirected* local multicast model considered in this paper. It follows the standard state machine based approach [1, 4, 5], similar to [9, 17]. Suppose there exists a set $F \subseteq V(G)$, of size at most f, such that G does not satisfy condition LCR with parameter F, but there exists an algorithm $\mathcal{A}$ that solves consensus on G. Algorithm $\mathcal{A}$ outlines a procedure $\mathcal{A}_u$ for each node u that describes u's state transitions, as well as messages transmitted on each channel of u in each round. Now there exists a graph $G' \in \Lambda_F(G)$ and a partition of $V(G')$ that does not satisfy the requirements of condition LCR. To create the required contradiction, we work with an algorithm for G' instead of $\mathcal{A}$. To see why this works, observe that an algorithm $\mathcal{A}$ on graph G can be adapted to create an algorithm $\mathcal{A}'$ for a graph $G' \in \Lambda_F(G)$ as follows. Each round i in the algorithm $\mathcal{A}$ is now split into two sub-rounds $i(a)$ and $i(b)$ in $\mathcal{A}'$. We consider each of these rounds separately and specify the corresponding steps for each node in G' for the algorithm $\mathcal{A}'$.

- *Round $i(a)$:* Each node $v \in V(G') \cap V(G)$ that was not split runs $\mathcal{A}_v$ as specified for round i. For a node $v \in V(G') - V(G)$ that was split into $v^0, v^1 \in V(G')$, both v^0 and v^1 run $\mathcal{A}_v$ for round i with the following modification. Consider a multicast channel $\chi_v \in \zeta_v$ of node v in G. Let χ'_{v^0} (resp. χ'_{v^1}) be the corresonding multicast channel in G' at node v^0 (resp. v^1). If the algorithm $\mathcal{A}_v$ wants to transmit a message on χ_v, then v^0 (resp. v^1) sends the message on χ'_{v^0} (resp. χ'_{v^1}), while v^1 (resp. v^0) ignores this message transmission. Observe that, for any node $u \in \chi_v$, u receives messages on the channel from exactly one of v^0 and v^1.

- *Round $i(b)$:* This round is reserved for the split nodes. Consider a node $v \in F - V(G')$ that was split into $v^0, v^1 \in V(G')$. Node v^0 forwards all messages it received in round $i(a)$ to v^1 and v^1 forwards all messages it received in round $i(a)$ to v^0. This allows both v^0 and v^1 to run $\mathcal{A}_v$ in the next round.

Now, $\mathcal{A}'$ might not solve consensus on G', or may not even terminate. However, as long as care is taken with regards to which nodes are allowed to be faulty in G' and the input of the split nodes, the guarantees for $\mathcal{A}$ will imply that $\mathcal{A}'$ does indeed terminate and solve consensus on G'. In particular, we want that

1. the faulty nodes in G' correspond to at most f nodes in G,
2. for each node $v \in F - V(G')$ that was split into $v^0, v^1 \in V(G')$, either
   a. both v^0 and v^1 have the same input, or
   b. at least one of v^0 and v^1 is faulty.

So for necessity, it is enough to show that no algorithm exists for a hypergraph $G' \in \Lambda_F(G)$, under the two conditions identified above. We formalize this property and use it in the formal necessity proof in [10].

7 Algorithm for the Local Multicast Model

To prove the sufficiency portion of Theorem 1, we work with a different network condition, which we will be equivalent to condition LCR. We first introduce some notation that is used in the algorithm. For a set of nodes $U \subseteq V(G)$, we use $G[U]$ to denote the subgraph induced by the nodes in U. The multicast channels in $G[U]$ are obtained from the multicast channels in G by removing nodes in $V(G) - U$ from each channel, with some channels possibly being deleted entirely. We use $G - U$ to denote the subgraph $G[V(G) - U]$.

A *path* is a sequence of distinct nodes such that if u precedes v in the sequence, then v is a neighbor of u in G (i.e., uv is an edge). For a path P and node z, we use $z \bullet P$ to denote the path obtained by prefixing the node z to P.

- *uv-paths:* For two nodes $u, v \in V(G)$, a uv-path P_{uv} is a path from u to v. u is called the *source* and v the *terminal* of P_{uv}. Any other node in P_{uv} is called an *internal* node of P_{uv}. Two uv-paths are *node-disjoint* if they do not share a common internal node.
- *Uv-paths:* For a set $U \subset V(G)$ and a node $v \notin U$, a Uv-path is a uv-path for some node $u \in U$. All Uv-paths have v as terminal. Two Uv-paths are *node-disjoint* if they do not have any nodes in common except the terminal node v. In particular, two node-disjoint Uv-paths have different source nodes. By definition, the number of disjoint Uv-paths is upper bounded by the size of the set U. Note the difference in definition between node-disjoint uv-paths and node-disjoint Uv-paths when $U = \{u\}$ is a singleton set. The former requires only internal nodes to be different, while the latter needs to have different source nodes as well. For the former, there can be more than one such node-disjoint path, while for the latter, there is at most one.
- *Propagate:* For two disjoint node sets $A, B \subseteq V(G)$, we use $A \rightsquigarrow_G B$ (read as A "propagates" to B in G) to denote that either
  (i) $B = \emptyset$, or
  (ii) for every $v \in B$, there exist at least $f+1$ node-disjoint Av-paths in the graph $G[A \cup B]$.

We now give a different network condition which is equivalent to condition LCR, but will be useful for specifying an algorithm for the local multicast model and proving its correctness. Recall that we use F' to denote the set of nodes in G' corresponding to nodes in F in G.

▶ **Definition 3.** *A graph G satisfies* condition AB with parameter F *if for every $G' \in \Lambda_F(G)$ and every partition (A, B) of $V(G')$, we have that either*

1. *$A \rightsquigarrow_{G'} B - F'$, or*
2. *$B \rightsquigarrow_{G'} A - F'$.*

We say that G satisfies condition AB*, if G satisfies condition AB with parameter F for every set $F \subseteq V$ of cardinality at most f.*

▶ **Theorem 4.** *A graph G satisfies condition LCR if and only if G satisfies condition AB.*

We skip the proof of Theorem 4, which is (almost) identical to proof of Theorem 5.2 in [9]. We show the sufficiency of condition AB (and hence condition LCR) constructively. For the rest of this section, we assume that G satisfies condition AB. The proposed algorithm is given in Algorithm 1. It draws inspiration from algorithms in [8, 9, 17]. Each node v maintains a binary state variable γ_v, which we call v's γ value. Each node v initializes γ_v to be its input value.

The nodes use "flooding" to communicate with the rest of the nodes. We refer the reader to [8, 9] for details about the flooding primitive. Briefly, when a node u wants to flood a binary value $b \in \{0, 1\}$, it transmits b to all of its neighbors, who forward it to their neighbors, and so forth. If a node u receives a message on channel χ, then u appends the channel id of χ when fowarding the message to its neighbors. By adding some simple sanity checks, one can assume that even a faulty node v does indeed transmit some value when it is v's turn to forward a message. In at most n synchronous rounds, the value b will be "flooded" in G. However, faulty nodes may tamper messages when forwarding, so some nodes may receive a value $\bar{b} \neq b$ along paths that contain faulty nodes.

The algorithm proceeds in phases. Every iteration of the `for` loop (starting at line 3) is a phase numbered $1, \ldots, 2^f$. Let F^* denote the actual set of faulty nodes. Each iteration of the for loop, i.e. phase > 0, considers a candidate faulty set F. In this iteration, nodes attempt to reach consensus, by updating their γ state variables, assuming the candidate set F is indeed faulty. Let Z and N be the set of nodes in $G - F$ that have their state variable set to 0 and 1, respectively, at the beginning of the iteration. Each iteration has three steps.

- In `step (a)`, each node v floods its γ_v value.
- In `step (b)`, based on the values received during flooding, each node v creates its estimate of the sets Z and N, by ignoring all paths that pass through the candidate faulty set F, i.e., paths that have internal nodes from F. This estimate is created in a manner so that
  1. when $F \neq F^*$, this estimate may be incorrect, but
  2. when $F = F^*$, this estimate is indeed correct.
- In `step (c)`, based on the estimates created in `step (b)`, a node v may update its γ_v value. The rules for updates ensure that
  1. when $F \neq F^*$, for each non-faulty node v, its state γ_v at the end of the iteration equals the γ value of some non-faulty node at the beginning of the iteration (Lemma 5).
  2. when $F = F^*$, all non-faulty nodes have identical γ values at the end of this iteration (Lemma 6).

Algorithm 1 Proposed algorithm for Byzantine consensus under the local multicast model: Steps performed by node v are shown here.

```
Each node has a binary input value in $\{0,1\}$.
Each node $v$ maintains a binary state $\gamma_v \in \{0,1\}$, initialized to the input value of $v$.
For each $F \subseteq V$ such that $|F| \leq f$ :
    Step (a): Flood value $\gamma_v$.
    if $v \in F$ then skip steps (b) and (c)

    Step (b):
    Create a graph $G'_v$ by splitting all nodes in $F$ as follows. Set
        $$F' := \{u^0 \mid u \in F\} \cup \{u^1 \mid u \in F\} \quad \text{and} \quad V(G'_v) := (V(G) - F) \cup F'.$$
     The edges and channels of $G'_v$ are as determined by the split operation, with the
     following choices: For every node $z \in F$ and a multicast channel $\chi_z \in \zeta_z$ :
        if $\exists w \in \chi_z$ such that $w \in V(G) - F$ then
            identify a single $wv$-path $P_{wv}$ in $G - F$ (Lemma 7).
            if $v$ received 0 from $z$ along the path $z \bullet P_{wv}$ in step (a), such that the initial
             message was sent by node $z$ on channel $\chi_z$ then assign $\chi_z$ to $z^0$.
            else assign $\chi_z$ to $z^1$.
        else
            assign $\chi_z$ to $z^1$.

    For each node $u \in V - F$, identify a single $uv$-path $P_{uv}$ in $G - F$ (Lemma 7). Note that
     path $P_{vv}$ trivially exists ($P_{vv}$ contains only $v$). Initialize a partition $(Z_v, N_v)$ of $V(G'_v)$
     as follows,
        $$Z_v := \{u^0 \mid u \in F\} \cup \{u \in V - F \mid v \text{ received value 0 along } P_{uv} \text{ in step (a)}\},$$
        $$N_v := \{u^1 \mid u \in F\} \cup (V - F - Z_v).$$

    Step (c):
    if $Z_v \rightsquigarrow_{G'_v} N_v - F$ then set $A_v := Z_v$ and $B_v := N_v$
    else set $A_v := N_v$ and $B_v := Z_v$

    if $v \in B_v - F$ then
        // by construction, the paths of interest in $G$ naturally correspond to
        paths in $G'_v$.
        if in step (a), $v$ received a value $\delta \in \{0,1\}$ identically along any $f+1$
         node-disjoint $A_v v$-paths in the graph $G'_v[A \cup (B - F)] = G'_v - (B_v \cap F)$ then
            set $\gamma_v := \delta$

Output $\gamma_v$
```

At the end, after all iterations of the main `for` loop, each output node v outputs its γ_v value.

The correctness of Algorithm 1 relies on the following two key lemmas, which are proven in Section A. Recall that we use F^* to denote the actual set of faulty nodes.

▶ **Lemma 5.** *For a non-faulty node $v \in V - F^*$, its state γ_v at the end of any given phase of Algorithm 1 equals the state of some non-faulty node at the start of that phase.*

▶ **Lemma 6.** *Consider a phase > 0 of Algorithm 1 wherein $F = F^*$. At the end of this phase, every pair of non-faulty nodes $u, v \in V - F^*$ have identical state, i.e., $\gamma_u = \gamma_v$.*

Lemma 5 ensures validity, i.e., that the output of each non-faulty node is an input of some non-fautly node. It also ensures that agreement among non-faulty nodes, once acheived, is not lost. Lemma 6 ensures that agreement is reached in at least one phase of the algorithm. These two lemmas imply correctness of Algorithm 1 as shown in Section A.

8 Conclusion

In this paper, we introduced the local multicast model which, to the best our knowledge, has not been studied before in the literature. The local multicast model encompasses the point-to-point, local broadcast, and hypergraph communication models, as well as some new models which have not been considered before. We identified a tight network condition for Byzantine consensus under the local multicast model, along the lines of [9, 17], and proved its necessity and sufficiency. When the local multicast model represents one of point-to-point, local broadcast, or hypergraph communication models, we showed how the identified network condition reduces to the known tight requirements for the corresponding case.

A natural extension to complete the local multicast model is to consider a *directed* communication graph, which corresponds to *directed* hypergraphs, and generalizes the directed cases of point-to-point and local broadcast models. In our recent work [10], we have extended the results in this paper to the directed setting. The natural extension of condition LCR to the directed case is the tight network condition for directed local multicast. We refer the reader to [10] for more details.

References

1 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., USA, 2004.

2 Vartika Bhandari and Nitin H. Vaidya. On reliable broadcast in a radio network. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 138–147, New York, NY, USA, 2005. ACM. doi:10.1145/1073814.1073841.

3 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *SIGOPS Oper. Syst. Rev.*, 41(6):189–204, 2007. doi:10.1145/1323293.1294280.

4 Danny Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

5 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, March 1986. doi:10.1007/BF01843568.

6 Mattias Fitzi and Ueli Maurer. From partial consistency to global broadcast. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 494–503, New York, NY, USA, 2000. ACM. doi:10.1145/335305.335363.

7 Alexander Jaffe, Thomas Moscibroda, and Siddhartha Sen. On the price of equivocation in byzantine agreement. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 309–318, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332491.

8 Muhammad Samir Khan, Syed Shalan Naqvi, and Nitin H. Vaidya. Exact Byzantine Consensus on Undirected Graphs under Local Broadcast Model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 327–336, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331619.

9 Muhammad Samir Khan, Lewis Tseng, and Nitin H. Vaidya. Exact Byzantine Consensus on Arbitrary Directed Graphs Under Local Broadcast Model. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.OPODIS.2019.30.

10 Muhammad Samir Khan and Nitin H. Vaidya. Byzantine consensus under directed hypergraphs. *arXiv*, 2021. arXiv:2109.01205.

11 Chiu-Yuen Koo. Broadcast in radio networks tolerating byzantine adversarial behavior. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 275–282, New York, NY, USA, 2004. ACM. doi:10.1145/1011767.1011807.

12 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

13 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

14 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

15 D. V. S. Ravikant, V. Muthuramakrishnan, V. Srikanth, K. Srinathan, and C. Pandu Rangan. On byzantine agreement over (2,3)-uniform hypergraphs. In *Distributed Computing*, pages 450–464, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

16 Lewis Tseng and Nitin Vaidya. Exact byzantine consensus in directed graphs. *arXiv preprint arXiv:1208.5075*, 2014. arXiv:1208.5075.

17 Lewis Tseng and Nitin H. Vaidya. Fault-tolerant consensus in directed graphs. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, page 451–460, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2767386.2767399.

A Proof of Correctness of Algorithm 1

In this section, we show correctness of Algorithm 1 when the communication graph G sastisfies condition AB. For the rest of this section, we assume that G satisfies condition AB. Throughout this section, we use F^* to denote the actual set of faulty nodes. We first prove Lemma 5.

Proof of Lemma 5. Fix a phase > 0. Note that a node updates its state only in `step (c)`. Suppose a node v updates its state γ_v to α. Then, as per the update rules in `step (c)`, v must have received the value α identically along $f+1$ node-disjoint $A_v v$-paths in `step (a)`. Since there are at most f faulty nodes, so at least one of these paths, say P, must have neither any faulty internal node nor a faulty source node. Since α was received along P, which has only non-faulty internal nodes, so the source node of P, say u, flooded α in `step (a)` of this phase. Since u is non-faulty, so γ_u had value α at the start of this phase. Therefore, the state of node v at the end of this phase equals the state of a non-faulty node u at the start of this phase. ◀

Before proving Lemma 9, we need some intermediate results. We first show that in every iteration of the main `for` loop, the paths in `step (b)` do exist.

▶ **Lemma 7.** *In any phase* > 0 *of the algorithm with a candidate faulty set* F*, for any two nodes* $u, v \in V(G) - F$*, there exists a* uv*-path in* $G - F$*.*

Proof. Suppose for the sake of contradiction that there exist two nodes $u, v \in V(G) - F$ such that there is no uv-path in $G - F$. Let A be the set of nodes that are reachable by node u in $G - F$, and let $B = V - A$. Note that

(i) $|F| \leq f$,

(ii) $u \in A = A - F$ so that $A - F \neq \emptyset$, and

(iii) $v \in B - F$ so that $B - F \neq \emptyset$.

Now, there are no edges between A and $B - F$. Since $|F| \leq f$, so there are at most f node-disjoint Av-paths and at most f node-disjoint Bu-paths in graph G. Therefore, we have

1. $A \not\rightsquigarrow_G B - F$, and
2. $B \not\rightsquigarrow_G A - F$.

Since $G \in \Lambda_F(G)$, so condition AB is violated, a contradiction. ◀

When a non-faulty node wants to flood a value $b \in \{0, 1\}$, it sends a single value b on all of its multicast channels. But a faulty node might send different messages on different channels. Note however, that even a faulty node must send the exact same value on a single multicast channel.

▶ **Lemma 8.** *Consider a phase* > 0 *of Algorithm 1 wherein* $F = F^*$*. For any two non-faulty nodes* $u, v \in V(G) - F^*$*, we have* $G'_u = G'_v$ *in* `step (b)` *of this phase. Furthermore, if in* `step (a)` *of this phase, faulty node* $z \in F^*$ *transmitted* 0 *(resp.* 1*) on one of its channels* $\chi_z \in \zeta_z$*, such that* $\chi_z - F^*$ *is non-empty, then in* `step (b)` *of this phase* χ_z *is assigned to* z^0 *(resp.* z^1*) in* $G'_u = G'_v$*.*

Proof. Consider the phase where $F = F^*$ and any two non-faulty nodes $u, v \in V(G) - F^*$. Observe that the node set of the two graphs G'_u and G'_v are the same. For the edges and channels, by construction, it is sufficient to show that for any $z \in F^*$, the assignment of multicast channels to z^0 and z^1 in the split operation is the same in G'_u as in G'_v. Consider an arbitrary node $z \in F^*$ and a multicast channel $\chi_z \in \zeta_z$ at node z. There are two cases to consider:

- **Case 1:** There exists a node $w \in \chi_z$ such that $w \in V(G) - F^*$.
  Let w be any arbitrary such node. By Lemma 7, there exists a wu-path (resp. wv-path) in $G - F^*$. Let P_{wu} (resp. P_{wv}) be any arbitrary wu-path (resp. wv-path) identified by u (resp. v) in line 9. Note that P_{wu} (resp. P_{wv}) does not contain *any* faulty nodes. Therefore, a message transmitted by z on χ_z, is received by u (resp. u) along $z \bullet P_{wu}$ (resp. $z \bullet P_{wv}$) untampered. Therefore, in `step (a)`, if z transmitted 0 on channel χ_z, then u (resp. v) received value 0 from z along $z \bullet P_{wu}$ (resp. $z \bullet P_{wv}$). So, in line 11, node u (resp. node v) assigns χ_z to z^0 in G'_u (resp. G'_v). Similarly, if z transmitted 1 on channel χ_z in `step (a)`, then both u and v assign χ_z to z^1 in G'_u and G'_v, respectively.
- **Case 2:** There does not exist any node $w \in \chi_z$ such that $w \in V(G) - F^*$.
  In this case, in line 13, both u and v assign χ_z to z^1 in G'_u and G'_v, respectively.

In both cases, we have that the multicast channel χ_z was assigned identically by both u and v. As shown in Case 1, if z transmitted 0 (resp. 1) on χ_z and $\chi_z - F^*$ is non-empty, then χ_z was assigned to z^0 (resp. z^1) by both u and v, as required. ◀

▶ **Lemma 9.** *Consider a phase > 0 of Algorithm 1 wherein $F = F^*$. Let*

$$Z := \left\{u^0 \mid u \in F\right\} \cup \{w \in V(G) - F^* \mid w \textit{ flooded value } 0 \textit{ in } \texttt{step (a)} \textit{ of this phase}\}$$
$$N := \left\{u^1 \mid u \in F\right\} \cup \{w \in V(G) - F^* \mid w \textit{ flooded value } 1 \textit{ in } \texttt{step (a)} \textit{ of this phase}\}.$$

For any two non-faulty nodes $u, v \in V(G) - F^*$, we have $Z_u = Z_v$ and $N_u = N_v$ in `step (b)` of this phase.

Proof. Consider the phase where $F = F^*$ and any two non-faulty nodes $u, v \in V(G) - F^*$. We show that $Z \subseteq Z_v$ and $N \subseteq N_v$ (resp. $Z \subseteq Z_u$ and $N \subseteq N_u$). Since $Z \cup N = Z_u \cup N_u = Z_v \cup N_v$, it follows that $Z = Z_u = Z_v$ and $N = N_u = N_v$. For a node $w \in F^*$, the two split nodes w^0 and w^1 are assigned identically by both u and v. So consider an arbitrary node $w \in V(G) - F^* = (Z \cup N) - \left\{u^0, u^1 \mid u \in F^*\right\}$. Recall that we are considering the phase > 0 of the algorithm where $F = F^*$ is the actual set of faulty nodes. There are two cases to consider:

- **Case 1:** $w \in Z - \left\{u^0 \mid u \in F^*\right\}$, i.e., $w \notin F^*$ flooded 0 in `step (a)` of this phase.
  Let P_{wv} be the wv-path identified by v in `step (b)`. Note that P_{wv} is contained entirely in $G - F^*$ so that P_{wv} does not have any faulty nodes. It follows that, in `step (a)`, since w flooded value 0 so v received value 0 along P_{wv}. Therefore, in `step (b)`, v puts w in the set Z_v.
- **Case 2:** $w \in N - \left\{u^1 \mid u \in F^*\right\}$, i.e., $w \notin F^*$ flooded 1 in `step (a)` of this phase.
  Let P_{wv} be the wv-path identified by v in `step (b)`. Note that P_{wv} is contained entirely in $G - F^*$ so that P_{wv} does not have any faulty nodes. It follows that, in `step (a)`, since w flooded value 1 so v received value 1 along P_{wv}. Therefore, in `step (b)`, v puts w in the set N_v.

So we have that $Z \subseteq Z_v$ and $N \subseteq N_v$, as required. A symmetric argument gives $Z \subseteq Z_u$ and $N \subseteq N_u$. As argued before, this implies that $Z = Z_u = Z_v$ and $N = N_u = N_v$. ◀

We are now ready to prove Lemma 6.

Proof of Lemma 6. Consider the phase where $F = F^*$. Suppose $u, v \in V(G) - F^*$ are any two non-faulty nodes. By Lemma 9, we have $Z = Z_u = Z_v$ and $N = N_u = N_v$, where Z and N are as in the statement of Lemma 9. By Lemma 8, we have $G'_u = G'_v$, Let $G' = G'_u = G'_v$. We use F' to denote the set of nodes in G' corresponding to nodes in F^* in G.

We now show that all non-faulty nodes in $V(G) - F^*$ have identical state at the end of this phase. Consider `step (c)` of this phase. If either $Z - F^*$ or $N - F^*$ is empty, then all non-faulty nodes have identical state at the start of the phase and they do not update their state in `step (c)`. So suppose that both $Z - F^*$ and $N - F^*$ are non-empty. Observe that, at the start of `step (c)`, all nodes in $Z - F^*$ have identical state of 0, while all nodes in $N - F^*$ have identical state of 1. We show that in `step (c)` either all nodes in $Z - F^*$ update their state to 1, or all nodes in $N - F^*$ update their state to 0.

Note that $G' \in \Lambda_{F^*}(G)$. By condition AB, either $Z \rightsquigarrow_{G'} N - F'$ or $N \rightsquigarrow_{G'} Z - F'$. We consider each case as follows.

- **Case 1:** $Z \rightsquigarrow_{G'} N - F'$.
  Consider an arbitrary node $v \in (Z \cup N) - F'$. In `step (c)`, v sets $A_v = Z$ and $B_v = N$. If $v \in A_v - F' = Z - F'$, then v has state 0 at the start of this phase and does not update it in `step (c)`. So suppose that $v \in B_v - F' = N - F'$. Now, if in `step (a)` v received the value 0 identically along some $f + 1$ node-disjoint Zv-paths in $G' - (N \cap F')$, then v sets $\gamma_v = 0$ in `step (c)`. We show that such $f + 1$ node-disjoint Zv-paths do indeed exist. Since $Z \rightsquigarrow_{G'} N - F'$, so there exist $f + 1$ node-disjoint Zv-paths in $G' - (N \cap F')$. Without loss of generality, only the source nodes on these paths are from Z. For each such path, observe that only the source node, say $z \in Z$, can be faulty. If the source node z is faulty, then by Lemma 8, and construction of G' and Z, z sent the value 0 on the first channel on this path in `step (a)`. If z is non-faulty, then by construction of Z, z flooded value 0 in `step (a)`. Now all other nodes on the path are non-faulty, so v received value 0 along this path in `step (a)`. Therefore, v received value 0 identically along the $f + 1$ node-disjoint Zv-paths in `step (a)`, as required.
- **Case 2:** $Z \not\rightsquigarrow_{G'} N - F'$ so that $N \rightsquigarrow_{G'} Z - F'$ by condition AB.
  Consider an arbitrary node $v \in (Z \cup N) - F'$. In `step (c)`, v sets $A_v = N$ and $B_v = Z$. If $v \in A_v - F' = N - F'$, then v has state 1 at the start of this phase and does not update it in `step (c)`. So suppose that $v \in B_v - F' = Z - F'$. As in Case 1, since $N \rightsquigarrow_{G'} Z - F'$, so there exist $f + 1$ node-disjoint Nv-paths in $G' - (Z \cap F')$ such that v received the value 1 identically along these paths in `step (a)`. Therefore, v sets $\gamma_v = 1$ in `step (c)`, as required.

In both of the cases, all non-faulty nodes have identical state at the end of this phase, as required. ◀

Using Lemmas 5 and 6, we can now prove the sufficiency of condition AB. Recall that by Theorem 4, condition AB is equivalent to condition LCR. Thus this shows the reverse direction of Theorem 1.

Proof of Theorem 1 ($\Leftarrow$ direction). Algorithm 1 satisfies the *termination* condition because it terminates in finite time.

In one of the iterations of the main `for` loop, we have $F = F^*$, i.e., F is the actual set of faulty nodes. By Lemma 6, all non-faulty nodes have the same state at the end of this phase. By Lemma 5, these states remain unchanged in any subsequent phases. Therefore, all nodes output an identical state. So the algorithm satisfies the *agreement* condition.

At the start of phase 1, the state of each non-faulty node equals its own input. By inductively applying Lemma 5, we have that the state of a non-faulty node always equals the *input* of some non-fautly node, including in the last phase of the algorithm. So the output of each non-faulty node is an input of some non-faulty node, satisfying the *validity* condition. ◀

Singularly Near Optimal Leader Election in Asynchronous Networks

Shay Kutten
Faculty of Industrial Engineering and Management,
Technion – Israel Institute of Technology, Haifa, Israel

William K. Moses Jr.
Department of Computer Science, University of Houston, TX, USA

Gopal Pandurangan
Department of Computer Science, University of Houston, TX, USA

David Peleg
Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot, Israel

Abstract

This paper concerns designing distributed algorithms that are *singularly optimal*, i.e., algorithms that are *simultaneously* time and message *optimal*, for the fundamental leader election problem in *asynchronous* networks.

Kutten et al. (JACM 2015) presented a singularly near optimal randomized leader election algorithm for general *synchronous* networks that ran in $O(D)$ time and used $O(m \log n)$ messages (where D, m, and n are the network's diameter, number of edges and number of nodes, respectively) with high probability.[1] Both bounds are near optimal (up to a logarithmic factor), since $\Omega(D)$ and $\Omega(m)$ are the respective lower bounds for time and messages for leader election even for synchronous networks and even for (Monte-Carlo) randomized algorithms. On the other hand, for general asynchronous networks, leader election algorithms are only known that are either time or message optimal, but not both. Kutten et al. (DISC 2020) presented a randomized asynchronous leader election algorithm that is singularly near optimal for *complete networks*, but left open the problem for general networks.

This paper shows that singularly near optimal (up to polylogarithmic factors) bounds can be achieved for general *asynchronous* networks. We present a randomized singularly near optimal leader election algorithm that runs in $O(D + \log^2 n)$ time and $O(m \log^2 n)$ messages with high probability. Our result is the first known distributed leader election algorithm for asynchronous networks that is near optimal with respect to both time and message complexity and improves over a long line of results including the classical results of Gallager et al. (ACM TOPLAS, 1983), Peleg (JPDC, 1989), and Awerbuch (STOC, 89).

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases Leader election, Singular optimality, Randomized algorithms, Asynchronous networks, Arbitrary graphs

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.27

Related Version *Full Version*: `https://arxiv.org/abs/2108.02197` [30]

Funding *Shay Kutten*: This work was supported in part by the Bi-national Science Foundation (BSF) grant 2016419.
William K. Moses Jr.: Part of this work was done while the author was a postdoc at the Technion – Israel Institute of Technology in Israel. This work was supported in part by a Technion fellowship and in part by NSF grants, CCF1540512, IIS-1633720, CCF-1717075, and BSF grant 2016419.

[1] Throughout, "with high probability" means "with probability at least $1 - 1/n^c$, for constant c."

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 27; pp. 27:1–27:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Gopal Pandurangan: This work was supported in part by NSF grants CCF-1717075, CCF-1540512, IIS-1633720, and BSF grant 2016419.
David Peleg: This work was supported in part by the US-Israel Binational Science Foundation grant 2016732.

1 Introduction

Background and motivation. Trade-offs between resource bounds (typically time and space) form a major subject of study in classical theory of computation. In distributed computing, it is common to focus on two fundamental measures, the time and message complexity of a distributed network algorithm, and trade-offs between time and communication have been well studied. See, e.g., [1, 4, 6, 7] for early trade-offs. A question that arose more recently regarding various distributed problems is whether the problem admits an algorithm that is optimal in time and communication simultaneously. In [21, 40], such algorithms are called *singularly optimal* (or *singularly "near optimal"* for algorithms whose complexity is polylogarithmically worse than optimal). Such algorithms have been shown in recent years for minimum spanning tree, (approximate) shortest paths, leader election, and several other problems [14, 22, 31, 40].

All the above results were shown for *synchronous* networks, and *do not* apply to asynchronous networks. In particular, singularly near optimal randomized *synchronous* leader election algorithms were presented in [31]. These algorithms require $O(D)$ time and use $O(m \log n)$ messages with high probability (where D, n and m are the network's diameter, number of nodes and number of edges, respectively). Singular near optimality follows from the fact that $\Omega(D)$ and $\Omega(m)$ are lower bounds for time and messages for leader election even for synchronous networks and even for randomized algorithms [31]. These algorithms inherently rely on the synchronous communication, so an attempt to convert them to asynchronous networks would probably incur heavy cost overheads (see the discussion of synchronizers below). In fact, the question whether similar bounds can be achieved for general *asynchronous* networks was left open, although one can obtain algorithms that are separately time optimal [41] or message optimal [17].

A singularly near optimal randomized *asynchronous* leader election algorithm was recently presented in [29], but only for *complete* networks. It requires $O(n)$ messages and $O(\log^2 n)$ time, which is singularly optimal (up to logarithmic factors) since $\Omega(n)$ and $\Omega(1)$ are the respective message and time lower bounds for leader election in complete n-node networks. The question whether leader election in general networks admits an *asynchronous* singularly near optimal algorithm or exhibits an inherent *time-messages trade-off* was again left as an open problem. The algorithm in [29] heavily utilizes the special nature of the complete graph, so designing an asynchronous algorithm for general graphs requires additional tools and insights.

Leader election is a central and intensively studied problem in distributed computing. It captures the pivotal notion of symmetry breaking in contexts involving the entire network ("global algorithms"). Given a leader, many other problems become trivial. Consequently, a singularly optimal leader election algorithm may ease the design of a singularly optimal algorithms for many other tasks. In addition to its theoretical importance, leader election is used in multiple practical contexts. The literature is too rich to cover here, but see, for example [1, 10, 11, 17, 19, 31, 33, 35, 41, 46, 49].

In our setting, an *arbitrary* subset of nodes can *wake up spontaneously at arbitrary times* and start the election algorithm by sending messages over the network. The algorithm should terminate with a *unique* node v being elected as leader (where initially, all the nodes are in the same state, “not leader”) and the leader’s identity should be *known to all nodes.* Our goal in this paper is to design leader election algorithms in distributed *asynchronous* networks that are singularly near optimal.

Using synchronizers. One can convert a synchronous algorithm to work on an asynchronous network using a standard tool known as a *synchronizer* [4]; however, such a conversion typically increases substantially either the time or the message complexity or both. Moreover, there is usually a non-negligible cost associated with *constructing* such a synchronizer in the first place. For example, applying the simple α synchronizer (which does not require the a priori existence of a leader or a spanning tree) to the singularly optimal synchronous leader election algorithm of [31] yields an asynchronous algorithm with message complexity of $O(mD \log n)$ and time complexity of $O(D)$; this algorithm is not message optimal, especially for large diameter networks. Indeed, many prior works (see e.g., [9]), do construct efficient synchronizers that can achieve optimal conversion from synchronous to asynchronous algorithms with respect to both time and messages, but constructing the synchronizer itself requires a substantial preprocessing or initialization cost. For example, the message cost of the synchronizer protocol of [9] can be as high as $O(mn)$. Moreover, several synchronizer protocols, such as β and γ of [4] and that of [9], require the existence of a *leader* or *a spanning tree*; hence these synchronizers are not useful for designing leader election algorithms. For these reasons, designing singularly optimal algorithms is more challenging for asynchronous networks than for synchronous ones and requires new approaches.

Distributed Computing Model. We model a distributed network as an arbitrary undirected connected graph $G = (V, E)$, $|V| = n$, $|E| = m$, similar to the standard model of [1,6,17,27], except that, in addition, processors can access *private unbiased coins.* Nodes have only knowledge of themselves and do not have any knowledge of their neighbors and their identities (if any). This is the commonly used *clean network* or KT_0 model (see e.g., [8]).

Our algorithm does not require that nodes have unique identities; however, it requires that nodes have knowledge of n, the network size, or at least a constant factor approximation of n. We note that several prior algorithms for leader election require knowledge of n [2,6,47]. If nodes have unique identifiers, we assume that they are of size $O(\log n)$ bits.

We assume the standard *asynchronous* $\mathcal{CONGEST}$ communication model [42], where messages (each message is of $O(\log n)$ bits) sent over an edge incur unpredictable but finite delays, in an error-free and FIFO manner (i.e., messages will arrive in sequence). However, for the sake of time analysis, it is assumed that message takes *at most one time unit* to be delivered across an edge. As is usual, we assume that local computation within a node is instantaneous and free; however, our algorithm will involve only lightweight local computations.

We follow the standard timing and wake-up assumptions used in prior asynchronous protocols (see [1,17,48]). Nodes are initially asleep, and a node enters the execution when it is woken up by the environment or upon receiving messages from awakened neighbors. As usual, uncertainties in the environment can be modeled by means of an *adversary* that controls some of the execution parameters. Specifically, we assume an *adversarial wake up* model, where node wake-up times are scheduled by an adversary (who may decide to keep some nodes dormant). The time complexity is measured from the moment the first node wakes up. A node can also be woken up by receiving messages from other nodes. In addition to the wake-up schedule, the adversary also decides the time delay of each message. These decisions are done *adaptively*, i.e., when the adversary makes a decision to wake up a node or delay a message, it has access to the results of all previous coin flips. The above adversarial wakeup model should be contrasted with the weaker *simultaneous wake up* model, where all nodes are assumed to be awake at the beginning of computation; simultaneous wake up is typically assumed in the design of synchronous protocols (see e.g., [1,31,32]). In the asynchronous setting, once a node enters execution, it performs all the computations required of it by the algorithm, and sends out messages to neighbors as specified by the algorithm.

Our Contribution. The main question addressed in this paper is whether singularly optimal bounds for leader election can be achieved for general *asynchronous* networks. We answer this question in the affirmative and present a randomized singularly near optimal leader election algorithm that elects a leader with high probability, runs in $O(D + \log^2 n)$ time with high probability, and has message complexity $O(m \log^2 n)$ with high probability, where high probability is probability at least $1 - 1/n^c$, for constant c. Our algorithm even works in anonymous networks. To the best of our knowledge, this is the first known distributed leader election algorithm for asynchronous general networks that is near optimal with respect to both time and message complexity and improves over a long line of results (see Table 1) including the classical results of Gallager et al. [17], Peleg [41], and Awerbuch [6]. We refer to Table 1 for a comparison of message and time complexity bounds of leader election algorithms in asynchronous networks. It should be noted that none of the prior results achieve time and message bounds that are simultaneously close to optimal bounds (even within a $O(\text{polylog}\, n)$ factor) of $\Theta(D)$ (time) and $\Theta(m)$ (messages) respectively. We note our bounds are almost as good as those obtained for the synchronous model: the work of Kutten et al. [31] presented a $O(m \log n)$ messages (with high probability) and a $O(D)$ algorithm. It is open whether one can design a (tight) singularly optimal leader election algorithm that uses $O(m)$ messages and $O(D)$ time even for the *synchronous* setting.

The importance of having a singularly optimal (or near optimal) leader election is that it can serve as a basic building block in designing singularly (or near) optimal asynchronous algorithms for other fundamental problems such as MST and shortest paths. Currently, we are not aware of such algorithms for these problems in *asynchronous* networks (unlike synchronous networks [14,22,40]).

Our algorithm makes use of several elementary techniques, combined in a suitable way. In particular, it exploits the idea of using groups of referees as *quorums* in order to ensure mutual exclusion, an idea utilized in several papers, e.g. [12,20,29,32,45]. Traditionally, verifying that an entire quorum has been secured is achieved by counting the number of supporting referees. It should be noted, though, that such a counting process is problematic under the asynchronous communication model. This difficulty requires us to introduce some *slack* to the quorum sizes, and rely on it in order to ensure quorum intersection with high probability. Another interesting feature of this algorithm concerns the way it manages communication. Message transmissions are performed using *flood-based broadcasts*, even when the message is targeted at a *single* recipient (unlike most previous algorithms, where such messages are sent by unicast along a specific path). This is done since in an asynchronous network, paths defined by previous broadcasts might not be shortest, so using them later might prevent us from attaining a near diameter time. The obvious down side is that using broadcasts for transmitting individual messages is expensive in communication. Hence, one delicate technical point is how to maintain a tight cap on the overall number of wasteful broadcasts, in order to save on messages and on congestion. The key idea is to ensure that the number of active "speakers" (as opposed to passive "relays" who merely forward messages) during the entire execution is kept small (logarithmic in the network size).

Table 1 Comparison of leader election algorithms for a general graph with n nodes, m edges, and D diameter in asynchronous systems along with our contributions. Note that if the algorithm was deterministic, then it was required that nodes have unique IDs. Randomized solutions may not have such a requirement.

Paper	Message Complexity	Time Complexity	Type of Solution
Gallager et al. [17]	$O(m + n \log n)$	$O(n \log n)$	Deterministic
Lavallée and Lavault [34]*	$O(m + n \log n)$	$O(n \log \log(n/\varepsilon))$	Randomized
Chin and Ting [13]	$O(m + n \log n)$	$O(n \log^* n)$	Deterministic
Gafni [16]	$O(m + n \log n)$	$O(n \log^* n)$	Non-deterministic
Awerbuch [5], Faloutsos and Molle [15]**	$O(m + n \log n)$	$O(n)$	Deterministic
Schieber and Snir [47]	$O(m + n \log n)$	$O(n)$	Randomized
Afek and Matias [2]	$O(m)$	$O(n)$	Randomized
Peleg [41]	$O(mD)$	$O(D)$	Deterministic
Awerbuch [6]***	$O(m^{1+\varepsilon})$	$O(D^{1+\varepsilon})$	Deterministic
This paper	$O(m \log^2 n)$	$O(D + \log^2 n)$	Randomized

*They claim a *virtual* running time of $O(D' \log \log(n/\varepsilon))$, corresponding to an algorithmically constructed subgraph of diameter D' of the initial network. D' may be as large as n. $0 < \varepsilon < 1$.

**The algorithm of [15] is a corrected version of the one in [5].

***Here, ε can be any value > 0.

We first compare the technical contribution of this work to the known singularly near optimal *synchronous* algorithm of Kutten et al. [31] for *general graphs*. The task there is significantly easier since nodes know when to terminate: once a node stops receiving an echo, exactly one node (the one with the highest random rank) will know it is the leader. Moreover, in the synchronous setting, this happens after $O(D)$ rounds. In the asynchronous setting, this is not possible (unless some heavy message overhead is added, e.g., by using a synchronizer). This leads to technical challenges that we overcome utilizing randomization, broadcasts, and quorums. Randomization is used not only to reduce the number of messages (similar to the synchronous case), but more importantly to implement the quorums.

We also compare the technical contribution of this work to the known singularly near optimal *asynchronous* algorithm of Kutten et al. [29] for *complete graphs*, which uses similar techniques. One key change concerns the way candidates communicate with referees. In [29], this is done by sending messages directly, which is doable in complete graphs. In contrast, in the current algorithm the candidates must rely on broadcasts to send messages to referees. This change raises additional challenges and necessitates a major modification to the algorithm of [29]. Specifically, in that algorithm, each candidate selects, and hence knows, its referees in each phase (this is doable because the graph is complete). In contrast, in our algorithm for an arbitrary graph, a candidate does not know the referees (because referees are chosen independently of the candidate). As a result, it is necessary to keep an accurate estimate of the number of referees. We rely on this estimate to ensure the existence of exactly one leader, with high probability (if the estimate is too low, multiple nodes might become leaders; if it is too high, no one will become a leader).

Moreover, it should be stressed that the technique used in [29] for saving on messages cannot be used here. Therein, candidates compete with each other in *phases*, and get eliminated gradually, until a single candidate remains. Here, there does not appear to be a way for the candidate to save by sending information to only specific referees in a message optimal manner. Specifically, applying the algorithm of [29] on a general graph by replacing direct communication with broadcasts (and making no other changes) would result in a high communication cost of $\Omega(mn)$ messages and possibly a higher run time (due to congestion), compared to the performance of the current algorithm.

Related Work. Leader election has been very well-studied in distributed networks for many decades. Le Lann [33] first studied the problem in a ring network and the seminal paper of Gallager, Humblet, and Spira [17] studied it in general graphs. Since then, various algorithms and lower bounds are known in different models with synchronous/asynchronous communication and in networks of varying topologies, such as cycles, complete graphs, or arbitrary graphs. See, e.g., [25, 27, 31, 32, 35, 41, 46, 49] and the references therein.

Prior to this paper, there were no known singularly near optimal algorithms for *general asynchronous* networks, i.e., algorithms that take $\tilde{O}(m)$ messages and $\tilde{O}(D)$ time.[2][3] Gallager, Humblet, and Spira [17] presented a minimum weight spanning tree (MST) algorithm (also applicable for leader election) with message complexity $O(m + n \log n)$ and time complexity $O(n \log n)$; this is (essentially) message optimal [31] but not time optimal. Hence, further research concentrated on improving the time complexity of MST algorithms. The time complexity was first improved to $O(n \log \log(n/\varepsilon))$, $0 < \varepsilon < 1$, by Lavallée and Lavault [34], then to $O(n \log^* n)$ independently by Chin and Ting [13] and Gafni [16], and finally to $O(n)$ by Awerbuch [5] (see also [15]). Thus using MST algorithms for leader election in asynchronous networks does not yield the best possible time complexity of $O(D)$. Peleg's leader election algorithm [41], on the other hand, takes $O(D)$ time and uses $O(mD)$ messages; this is time optimal, but not message optimal.

Korach et al. [28], Humblet [23], Peterson [44] and Afek and Gafni [1] presented $O(n \log n)$ message algorithms for *asynchronous complete* networks. Korach, Kutten, and Moran [27] presented a general method plus applications to various classes of graphs.

For *anonymous* networks under some reasonable assumptions, deterministic leader election was shown to be impossible, using symmetry arguments [3]. Randomization comes to the rescue in this case; random rank assignment is often used to assign unique identifiers, as done herein. Randomization also allows us to beat the lower bounds for deterministic algorithms, albeit at the risk of a small chance of error. As a starting step, Schieber and Snir [47] developed a randomized algorithm for leader election in anonymous asynchronous networks that took $O(m + n \log n)$ messages and $O(n)$ time. Afek and Matias [2] presented a randomized algorithm that took the same time but improved the message complexity to $O(m)$, which is message optimal.[4]

For synchronous networks, Pandurangan et al. [40] and Elkin [14] have presented singularly near optimal distributed algorithms for MST. Note that optimal time for MST means $\Omega(D + \sqrt{n})$ [43], while for leader election in asynchronous networks, optimal time is $O(D)$, as shown in [41] but using worse message complexity.

We note that the singularly optimal algorithm of this paper (for asynchronous networks) as well as those of [40] and [14] (for synchronous networks) assume the so-called *clean network model*, a.k.a. KT_0 [42] (see Section 1), where nodes do not have initial knowledge of the identity of their neighbors. But the above optimal results do not in general apply to the KT_1 model, where nodes have initial knowledge of the identities of their neighbors. Clearly, the distinction between KT_0 and KT_1 has no bearing on the asymptotic bounds for the time complexity, but it is significant when considering message complexity. Awerbuch et al. [8] show that $\Omega(m)$ is a message lower bound for broadcast (and hence for leader

[2] $\tilde{O}$ notation hides a polylog(n) factor.

[3] There was, however, work done on a ring where Itai and Rodeh [24] presented an algorithm with $O(n \log n)$ bit complexity on expectation. Further analysis shows that the message complexity is $O(n \log n)$ on expectation and the time complexity is $O(n \log n)$ on expectation.

[4] Note that [2]'s message complexity is $O(m)$ when the solution succeeds with constant probability. For a solution that succeeds with high probability, the message complexity grows to $O(mn \log^2 n)$.

election and MST) in the KT_1 model, if one allows only (possibly randomized Monte Carlo) comparison-based algorithms, i.e., algorithms that can operate on IDs only by comparing them. (We note that all algorithms mentioned earlier are comparison-based, including ours.) Hence, the result of [8] implies that our leader election algorithm (which is comparison-based and randomized) is *time* and *message near optimal* in the KT_1 model if one considers comparison-based algorithms only.

On the other hand, for *randomized non-comparison-based* algorithms, the message lower bound of $\Omega(m)$ does not apply in the KT_1 model. King et al. [26] presented a randomized, non-comparison-based Monte Carlo algorithm in the KT_1 model for spanning tree, MST (and hence leader election) in $\tilde{O}(n)$ messages ($\Omega(n)$ is a message lower bound) and in $\tilde{O}(n)$ time (see also [36]). While this algorithm shows that one can achieve $o(m)$ message complexity (when $m = \omega(n \operatorname{polylog} n)$), it is *not* time-optimal; the time complexity for spanning tree and leader election is $\tilde{O}(n)$. The works of [18, 21] showed bounds with with improved round complexity, but with worse bounds on the message complexity and more generally, trade-offs between time and messages [21]. We note that all these results are for *synchronous* networks. For *asynchronous* networks, Mashreghi and King [37, 38] presented a spanning tree algorithm (also applies for leader election and MST) that takes $\tilde{O}(n^{1.5})$ messages and $\tilde{O}(n)$ time. It is an open question whether one can design a randomized (non-comparison based) algorithm that takes $\tilde{O}(n)$ messages and $\tilde{O}(D)$ rounds in the KT_1 model; these are the optimal bounds possible in the KT_1 model.

2 A Singularly Optimal Asynchronous Leader Election Algorithm

In this section, we present a leader election algorithm in asynchronous networks that is essentially optimal with respect to both message and time complexity. We assume that all nodes have knowledge of n, the number of nodes in the network.

2.1 Algorithm

Brief Overview. The process is initiated by one or more nodes, which are woken up by the adversary, possibly at different times (see earlier discussion of the adversarial wakeup model in Section 1). These nodes wake up the rest of the nodes. Subsequently, each awake node decides whether it will participate in the algorithm in the role of (i) a candidate and/or (ii) a referee. To do that, the node chooses randomly (for each role separately), with probability $O(\log n/n)$, whether to take on the role or not.[5] Each *candidate* attempts to become the leader by winning over a sufficient number of referees. *Referees* are used to decide which candidate will go on to become the leader, essentially preferring a stronger candidate (one who randomly chose a higher rank) over a weaker (lower rank) one, provided the stronger candidate does not "show up too late" (namely, after the weaker candidate has already accumulated sufficiently many referees to declare itself leader). Once a candidate becomes a leader, it broadcasts the message that it is leader and that all nodes should terminate.

Detailed Description. Each awake node u maintains the following information: (i) a rank $RANK_u$, chosen uniformly at random from $[1, n^4]$ (we assume that all nodes have knowledge of n, the network size), (ii) two indicator variables CAND-STATE and REF-STATE reflecting the status of its candidacy for leadership and its role as a referee, respectively, and (iii)

[5] It is possible for a node to participate in both roles or not participate in either.

the set M-List of all messages u has heard over the course of the algorithm.[6] The variable CAND-STATE can take one of three values, Candidate, Non-elected, and Elected, denoting whether the node is a candidate for leadership, it irrevocably committed itself to not be a leader, or it irrevocably committed itself to be a leader, respectively. The variable REF-STATE can take one of four values Non-selected, Ready, Chosen-Selected, and In-Dispute, with each of the states described in detail later on.

Initially, a node u is asleep and may be awoken by either the adversary or a $\langle$WAKEUP$\rangle$ message that originated at another node. (For the sake of uniformity, it is convenient to think of the the adversary waking up a node as a WAKEUP message arriving from outside the system, and treat both events in the same way.) Once awoken, u calls Procedure **Initialize**, which first wakes up adjacent nodes by sending the message $\langle$WAKEUP$\rangle$ to u's neighbors.[7] Subsequently, u chooses randomly whether or not to be a candidate by flipping a biased coin with probability $1000 \log n/n$: if successful, u initializes CAND-STATE to Candidate, chooses a rank at random - an integer in $[1, n^4]$, and broadcasts a request carrying its rank.[8] If not successful, u sets CAND-STATE to Non-elected. Node u also chooses randomly whether or not to be a referee, with probability $1000 \log n/n$: if successful, u initializes REF-STATE to Ready and sets the variables CONTENDER and CHOSEN to -1, else it sets REF-STATE to Non-selected. The pseudocode for the initialization procedure, as well as for others, is present in the full version of the paper.

Consider some node v. Any message that v wants to send, whether to continue a previous broadcast or to pass on a newly generated message, is added to its list Send-List(e) for each of its edges e. Whenever one of v's outgoing edges e is free for a new message to be sent across it, v invokes Procedure **Send_Message** for that edge, which picks an arbitrary unsent message from the list Send-List(e), transmits it over the edge, and erases it from the list. Any messages that were generated by the node are added also to M-List, a list of messages the node has already heard.

If node u becomes a candidate (respectively, a referee), then its subsequent actions in this role are governed by Procedure **Candidate** (resp., Procedure **Referee**).[9] The appropriate procedure between these two procedures is called when the node receives a message. Another node's leader announcement message is processed outside of these two procedures. Additionally, when a relevant message is received, the Procedure **Candidate_Dispute_Response** may also be invoked to resolve a dispute (to be described later). In addition to whichever procedure is called, if any, every awake node stores all the messages it receives in its list M-List and participates in every broadcast that reaches it by also adding the message to Send-List(e) for all its edges e except those over which the message was received.[10] A node u, once awake, will eventually either broadcast a LEADER message (in its role as a candidate) and terminate, or receive such a message about another candidate, at which point u stores the rank of the leader and then terminates (having also forwarded the transmitted message to its neighbors). The process governing the responses of the nodes to different incoming messages is called **On_Receive_Message**.

[6] Actually, our algorithm can be extended if nodes have a knowledge of n that is within some (known) constant factor. We discuss this at the end of Section 2.2.

[7] Here, and in other places, we say that a node u sends messages only to its neighbors. This is the local description of the algorithm. Globally, this results in the message being broadcast throughout the graph.

[8] The theorems in this paper hold for all sufficiently large n, say $n \geqslant n_0$, where the value of n_0 depends on the probability used for the coin flip. The particular value of $1000 \log n/n$ was chosen for ease of exposition, but it can be modified in order to yield a smaller n_0. No attempt was made to optimize this value.

[9] Recall that these two procedures may run concurrently in the same node, in case it assumed both roles.

[10] Note that if a message currently in v's Send-List(e) is received over edge e, then v removes that message from Send-List(e).

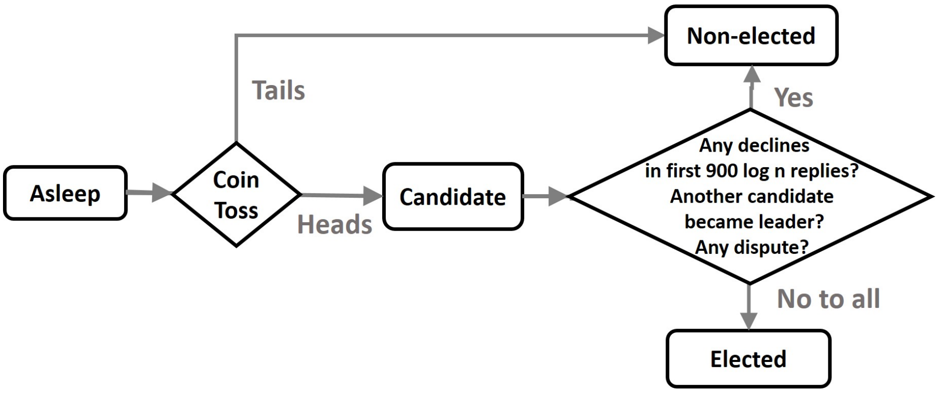

Non-elected
Tails
Yes
Asleep
Coin Toss
Candidate
Heads
Any declines in first 900 log n replies? Another candidate became leader? Any dispute?
No to all
Elected

Figure 1 The state progression of a candidate.

Procedure **Candidate** is run by a candidate to help it determine whether it will become the leader or not. Each candidate u broadcasts a request message during Procedure **Initialize**. Candidate u then waits for $900 \log n$ replies from different referees.[11] If one of these replies is a decline message, i.e., a referee says that u is declined from being the leader, then u updates its CAND-STATE to Non-elected and sends the message $\langle RANK_u, \text{LOSES}\rangle$ to all neighbors. Otherwise if all the replies are APPROVED messages (and u has not terminated yet), u becomes the leader, i.e., it changes its CAND-STATE to Elected, then announces this to all the nodes and terminates.[12] The state progression of a candidate is pictorially represented in Figure 1.

Procedure **Referee** is run by a referee r to decide which candidate becomes the leader. A referee's REF-STATE may be set to one of the following three values.

- REF-STATE = Ready holds when r has not heard yet of any candidate.
- REF-STATE = Chosen-Selected holds when r has approved one candidate, referred to as its *chosen* candidate, and has declined every other candidate that approached it so far. The rank of the chosen candidate is stored in the variable CHOSEN.
- REF-STATE = In-Dispute holds when r currently keeps track of two candidates: the *chosen* v, whose rank $RANK_v$ is stored in the variable CHOSEN, and a *contender* w, whose rank $RANK_w$ is stored in the variable CONTENDER, such that $RANK_w > RANK_v$, and all other candidates that approached r so far were declined. In such a case, a *dispute* is currently in progress between v and w.

The state In-Dispute is typically reached when r has a chosen candidate v that was approved by it, and later it gets a request from another candidate w such that $RANK_w > RANK_v$. In this situation, r cannot decline w (since it has a higher $RANK$ than its current chosen), but at the same time, it cannot approve w, since it may be that v had already collected enough approvals and became the leader in the meantime. To resolve this uncertainty, r declares a dispute, and broadcasts a message containing the dispute information, $\langle RANK_v, RANK_w, \text{DISPUTE}\rangle$, so that when v eventually receives the message, it can make the comparison between itself and w and resolve the dispute (v wins iff it already became the leader prior to receiving the dispute message). While waiting for v's response, r stores $RANK_w$ of w in the variable CONTENDER.

[11] A candidate may receive more than $900 \log n$ replies as there may be more referees. In such a case, the candidate only considers the *first* $900 \log n$ replies it receives.

[12] Note that messages about another candidate becoming the leader (and thus possibly affecting u's candidacy) are not handled in this procedure but are instead handled in Procedure **On_Receive_Message**.

As can be seen, the referee changes state only as a consequence of receiving a request from a candidate or receiving the result of a dispute from its chosen candidate.

We now observe how a referee r responds upon receiving a message $\langle RANK_u, \textsc{request}\rangle$ from a candidate u. We only consider what happens if r's REF-STATE $\neq$ Non-selected as otherwise r is not a referee. Three states are possible.

(A) If r's REF-STATE = Ready, meaning that u is the first contender approaching r with a request, then r registers u as its chosen candidate by storing $RANK_u$ in the variable CHOSEN, broadcasts an approval message for u, $\langle RANK_u, RANK_r, \textsc{approved}\rangle$, and switches its REF-STATE to Chosen-Selected.

(B) If r's REF-STATE = Chosen-Selected, and the current chosen candidate is v, then there are two possibilities. The simpler situation is when v is stronger than u, i.e., $RANK_v > RANK_u$, in which case r may immediately broadcast a decline message for u, $\langle RANK_u, RANK_r, \textsc{declined}\rangle$.

The other case is that u is the stronger of the two candidates, i.e., $RANK_v < RANK_u$. In this case, u should normally replace v as the chosen candidate, except if v has already declared itself leader in the meantime. The way to resolve this question is a dispute between u and v. First, r checks its list M-List of previously received messages to see if such a dispute between u and v is already in progress, i.e., if M-List contains a previously received message of the form $\langle RANK_v, RANK_u, \textsc{dispute}\rangle$ announcing the initiation of a dispute between u and v, or even a message of the form $\langle RANK_v, \textsc{loses}\rangle$ announcing the outcome of such a dispute (such a message necessarily indicates that v has *lost* the dispute, since a "win" by v can only occur if v has already declared itself leader, in which case v has already broadcast this fact and hence need not reply to the dispute).[13] There are three possible situations.

- Referee r has already received a message with the outcome of the dispute (namely, v lost). Then r only updates CHOSEN to $RANK_u$ and broadcasts an approval message for u, $\langle RANK_u, RANK_r, \textsc{approved}\rangle$.
- Referee r has received a message announcing a dispute, but has not yet heard about the outcome. Then r only updates CONTENDER to $RANK_u$ and updates REF-STATE to In-Dispute and awaits news on the outcome (but does not broadcast any new messages).
- Referee r did not hear of an existing dispute between u and v. Then it is up to r to initiate a dispute, so r registers u as the contender by storing $RANK_u$ in the variable CONTENDER, sets its REF-STATE to In-Dispute, and broadcasts a dispute message for v of the form $\langle RANK_v, RANK_u, \textsc{dispute}\rangle$.

(C) If r's REF-STATE = In-Dispute, signifying that *another* dispute (between v and some other candidate) is in progress, then r compares the new candidate u with the current contender w. If u is weaker than w ($RANK_u < RANK_w$), then r immediately broadcasts a decline message for u, $\langle RANK_u, RANK_r, \textsc{declined}\rangle$. Otherwise ($RANK_u > RANK_w$), r broadcasts a decline message for w, $\langle RANK_w, RANK_r, \textsc{declined}\rangle$, updates CONTENDER to $RANK_u$, and initiates a new dispute by broadcasting a dispute message for CHOSEN of the form $\langle \mathsf{CHOSEN}, RANK_u, \textsc{dispute}\rangle$.

The procedure to handle the referee's actions on receiving a request is called Procedure **Referee_Request_Response**.

[13] Note that a $\langle RANK_v, \textsc{loses}\rangle$ message may be the result of a dispute between a candidate v and some other candidate, not necessarily u. It is sufficient that v lost its candidacy.

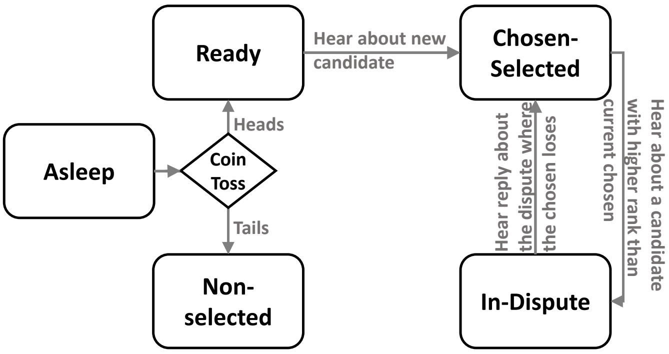

Ready
Hear about new candidate
Chosen-Selected
Heads
Asleep
Coin Toss
Tails
Non-selected
Hear reply about the dispute where the chosen loses
Hear about a candidate with higher rank than current chosen
In-Dispute

Figure 2 The state progression of a referee.

Finally, let us describe how a node v which is currently the chosen candidate of some referee r (but may have possibly changed its CAND-STATE since the time it was approved by r) handles a dispute request. Notice that a $\langle RANK_v, RANK_u, \text{DISPUTE}\rangle$ message is only sent to v when it is weaker than u ($RANK_v < RANK_u$), so when v receives such a message, it must abdicate its candidacy (if it is still a candidate), unless it has already elected itself as leader (which is an irreversible decision) and terminated. Hence, if v's CAND-STATE = Candidate, then v relinquishes its candidacy by setting CAND-STATE to Non-elected and broadcasts the result of the dispute as $\langle RANK_v, \text{LOSES}\rangle$.[14] If, however, v's CAND-STATE is set to Elected, then v has already broadcast a leader announcement message and terminated, so no additional response to the dispute message is required. The procedure to handle the actions of the chosen candidate on receiving a dispute message from a referee is called Procedure **Candidate_Dispute_Response**.

As there are multiple referees generating messages for various disputes, a referee may receive the results of a dispute it does not need to immediately process (but the result is stored for future processing, if any). If r's REF-STATE = In-Dispute, r's chosen is v, r's contender is u, and r receives a reply to a dispute of the form $\langle RANK_v, \text{LOSES}\rangle$, then r immediately processes the message as follows.[15] First, r updates CHOSEN to $RANK_u$, sets CONTENDER to -1, and updates REF-STATE to Chosen-Selected. Subsequently, r initiates the broadcast of an approval message for u, $\langle RANK_u, RANK_r, \text{APPROVED}\rangle$. The procedure to handle the referee's actions on receiving a reply from the chosen about an ongoing dispute is called Procedure **Referee_Dispute_Reply_Response**. The state progression of a referee is seen in Figure 2.

2.2 Analysis

We now prove the correctness of the algorithm and analyze its complexity. We prove a weaker time complexity bound of $O(D\log^2 n)$ here, with the stronger result of $O(D + \log^2 n)$ in Section 2.3.2. Before getting into the meat of the proof, let us make an important observation and subsequently state a useful lemma.

[14] Note that this broadcast operation is unnecessary when v's CAND-STATE = Non-elected, as v would have previously broadcast a message announcing its loss. This broadcast would have been the result of either another dispute involving v or v receiving a decline from one of the referees.

[15] Note that the message $\langle RANK_v, \text{LOSES}\rangle$ may also be generated by a candidate v upon receiving a decline message from a referee. For ease of writing, when we say "reply to a dispute", we mean any message of the form $\langle RANK_v, \text{LOSES}\rangle$, regardless of how it was generated.

▶ **Observation 1.** *From the time the first node is awake, all nodes are awakened in $O(D)$ time using $O(m)$ messages.*

Set $\mathcal{R}_\ell = 900 \log n$ and $\mathcal{R}_h = 1100 \log n$. By Observation 1, we see that all nodes awaken and thus participate in candidate selection and referee selection. Denote by N_C and N_R the number of candidates and referees selected in the algorithm, respectively. We now bound N_C and N_R with high probability.

▶ **Lemma 2.** *With probability $1 - 1/n^3$, both the number of candidates N_C and the number of referees N_R are in $[\mathcal{R}_\ell, \mathcal{R}_h]$.*

Proof. The random choice of a single candidate can be viewed as a Bernoulli trial with probability $1000 \log n/n$. Thus, the total number of candidates chosen N_C is the sum of n independent Bernoulli trials. Using known Chernoff bounds such as Theorem 4.4 and 4.5 in [39] where $\delta = 1/10$ and $\mu = 1000 \log n$, we see that the bounds in the lemma hold with the required probability. A similar argument holds for N_R. ◀

We are now ready to argue the correctness of the algorithm, i.e., show that exactly one candidate becomes a leader with high probability. This is done in two stages, showing first that the number of candidates that become leaders is *at least* one and then that this number is *at most* one. Throughout the following lemmas, we require that each node has a unique $RANK$. It is easy to see that this is true with high probability since each node selects its rank uniformly at random from $[1, n^4]$.

▶ **Lemma 3.** *At least one candidate becomes a leader with high probability.*

Proof. By Lemma 2, at least one node becomes a candidate with high probability and chooses a RANK. Each candidate waits for replies from $\mathcal{R}_\ell$ referees before deciding to become a leader. By Lemma 2, at least that many nodes become referees with high probability. Thus, there exist enough referees with high probability that generate replies for a candidate so that it can become a leader.

Let u be the candidate with the highest RANK. Now u broadcasts its candidacy to all referees. Either u wins at every referee, receives the responses, and becomes a leader. Or else, u loses at some referee, which implies that some other candidate is a leader. Thus at least one candidate becomes a leader. ◀

▶ **Lemma 4.** *At most one candidate becomes a leader with high probability.*

Proof. Each candidate waits for positive replies from $\mathcal{R}_\ell$ referees in order to become a leader. By Lemma 2, with high probability, the number of referees that exist in the system satisfies

$$N_R \geqslant \mathcal{R}_\ell > 0.8 N_R.$$

Put another way, given that $N_R \in [\mathcal{R}_\ell, \mathcal{R}_h]$, for any two candidates u and v that receive replies from the sets of referees R_u and R_v in order to decide on becoming a leader,

$$\begin{aligned} |R_u \cap R_v| &= |R_u \cup R_v| - |R_u \setminus R_v| - |R_v \setminus R_u| \geqslant \mathcal{R}_\ell - (\mathcal{R}_h - \mathcal{R}_\ell) - (\mathcal{R}_h - \mathcal{R}_\ell) \\ &= 900 \log n - (1100 \log n - 900 \log n) - (1100 \log n - 900 \log n) > 1. \end{aligned}$$

We show that when two candidates u and v share a referee r whose replies help them determine if they may become a leader, it is impossible for both candidates to become leaders. Thus, no more than one candidate becomes a leader with high probability. Without loss of generality, let $RANK_u < RANK_v$. Consider the sequence of arrival of candidacy messages at r and replies. The following three cases cover all possible scenarios.

Case 1: r knows of a candidate w where either $RANK_w > RANK_u$ or w has become a leader before a dispute message generated by r reaches it.

If r knows of a candidate w where $RANK_w > RANK_u$, then it is clear that at least u will be rejected and not become the leader. Otherwise, if r receives a candidacy message from either u or v, r will generate a dispute message and send it to w. If w has become the leader before this dispute message reaches it, then w would have already generated a leader announcement message and terminated. Node r meanwhile will not confirm u or v as a leader until it hears back from w. So whichever of u's or v's candidacy message was at r will not be approved once w's leader announcement message reaches node r and both candidates will not become the leader.

In this situation, it is guaranteed that at least one of u or v not become the leader.

Case 2: v's candidacy message reaches r first and r subsequently replies that v may become the leader, all before u's candidacy message reaches r.

In this case, since $RANK_u < RANK_v$, r declines u once its candidacy message reaches r. It is also possible that v may have become the leader and generated a leader announcement message. In this situation, that message may propagate to r and u, also resulting in u not becoming the leader. In either situation, u will not become the leader.

Case 3: u's candidacy message reaches r first and r subsequently replies that u may become the leader, all before v's candidacy message reaches r.

In this case, if u received enough approvals and became the leader, then it may generate a leader announcement message. Now, this leader announcement message will either reach r before or after v's candidacy message reaches it. In either case, v will not become the leader because r will not approve v before receiving the result of the dispute from u. If however, u did not receive enough approvals before v's candidacy message reaches r and r's subsequently generated dispute reaches u, then u will give up its candidacy.

In either case, at most one of u or v will become the leader (perhaps neither of them). ◀

Thus, the algorithm is correct. We now give a useful lemma that is subsequently used to bound both the message and time complexity of the algorithm.

▶ **Lemma 5.** *Any node generates at most $O(\log n)$ unique messages with high probability to be broadcast over the course of the algorithm.*

Proof. There is one instance of a WAKEUP message sent through the system. In addition, each candidate generates one candidacy request message and possibly one leader announcement message or candidacy loss message (as a reply to a dispute). Thus each candidate generates $O(1)$ messages. Each referee generates a reply to each candidate it hears from and possibly a dispute message with the current chosen as well. By Lemma 2, there are $O(\log n)$ candidates a referee may have to reply to and generate disputes for, resulting in each referee generating $O(\log n)$ messages. ◀

▶ **Lemma 6.** *There are $O(\log^2 n)$ unique messages with high probability broadcast in the system over the course of the algorithm.*

Proof. Aside from the initial WAKEUP message, only candidates and referees generate unique messages to be broadcast. There are $O(\log n)$ such candidates and referees with high probability by Lemma 2. Thus, there are totally $O(\log^2 n)$ unique messages with high probability broadcast in the system over the course of the algorithm. ◀

Combining the lemma with the fact that each broadcast of a unique message results in $O(m)$ messages, we get the following.

▶ **Corollary 7.** *The total message complexity is $O(m \log^2 n)$ with high probability.*

▶ **Lemma 8.** *The run time of the algorithm is $O(D \log^2 n)$ with high probability.*

Proof. By Observation 1, all nodes wake up in $O(D)$ time. One of the woken nodes, say u, will go on to become the leader. Let us bound the number of "logical phases" of broadcasts needed until all the nodes are made aware that u is the leader. (Note that we only use the word "phase" in this analysis, but we do not use this terminology in the algorithm itself.) Each phase is responsible for certain information being broadcast to nodes, and the phases, as described, occur sequentially. Furthermore, different phases may take different amounts of time. In the first phase, u broadcasts its candidacy. In the next phase, each of the referees may need to broadcast a dispute. In the subsequent phase, each of the candidates that is a target of a dispute needs to broadcast its reply. In the next phase, each of these referees broadcasts its reply to u's candidacy request. In the final phase, u broadcasts that it is the leader. Thus, there are $O(1)$ such phases.

In each of these phases, a broadcast originating at some node u is complete when the message m reaches all other nodes. The shortest path between u and any other node is of length at most $O(D)$. By Lemma 6, there are at most $O(\log^2 n)$ unique messages generated in the system with high probability. Thus m may be delayed at each node in the shortest path by at most $O(\log^2 n)$ other messages with high probability, resulting in a total time of $O(D \log^2 n)$ with high probability for the phase to complete. Since there are $O(1)$ phases, the total time until the algorithm completes is $O(D \log^2 n)$ with high probability. ◀

▶ **Theorem 9.** *There exists an algorithm that solves leader election with high probability in any arbitrary graph with n nodes, m edges, and diameter D in $O(D \log^2 n)$ time with high probability using $O(m \log^2 n)$ messages with high probability in an asynchronous system with adversarial node wakeup.*

2.3 Improvements

2.3.1 Knowledge of n

In the above analysis, it may be noted that nodes do not need to know the exact value of n. In fact, it is easy to extend the algorithm and analysis if nodes know the value of n up to a constant factor. More precisely, it is sufficient if all nodes know either (i) the value of a constant c_1, where $0 < c_1 \leqslant 1$, and a lower bound on n, n', such that $c_1 n \leqslant n' \leqslant n$ or (ii) the value of a constant c_2, where $1 \leqslant c_2$, and an upper bound on n, n^*, such that $n \leqslant n^* \leqslant c_2 n$. By adjusting the coin toss probability to some $c \log n'/n'$ (or $c \log n^*/n^*$) for a carefully chosen value of c, we can show that the analysis goes through for a sufficiently large n.

2.3.2 Reducing Time Complexity to $O(D + \log^2 n)$

The analysis of the runtime in Section 2.2 can be tightened further. The below lemma comes from Theorem 1 in Topkis [50], adapted to the current setting and terminology. Notice that we can use their Theorem because the process of flooding they study is being implemented here via **Send_Message**(e).

▶ **Lemma 10.** *It takes $D + k - 1$ time to broadcast k messages in a graph with diameter D in the asynchronous setting.*

Combining Lemma 10 with Lemma 6, which states that there are at most $O(\log^2 n)$ unique messages with high probability, and the argument (from the proof in Section 2.2) that there are $O(1)$ "logical phases" of broadcasts, we see that the run time of the algorithm is $O(D + \log^2 n)$ with high probability. When coupled with our previous analysis of correctness and message complexity, we arrive at the following theorem.

▶ **Theorem 11.** *There exists an algorithm that solves leader election with high probability in any arbitrary graph with n nodes, m edges, and diameter D in $O(D + \log^2 n)$ time with high probability using $O(m \log^2 n)$ messages with high probability in an asynchronous system with adversarial node wakeup.*

3 Conclusion

We have presented a randomized algorithm for asynchronous leader election in general networks. Our algorithm has message and time bounds that are both within a polylogarithmic factor of the lower bound, and is the first such singularly optimal algorithm presented for general asynchronous networks.

Two important open questions remain. First, is it possible to obtain near singularly optimal bounds using a *deterministic* algorithm? Our algorithm needs an accurate knowledge of the network size (at least up to a constant factor). Second, can we get a (near) singularly optimal algorithm (even randomized) without the restriction that nodes need an accurate knowledge of n or even no knowledge of n? Third, can we design (near) singularly optimal algorithms for other fundamental problems such as minimum spanning tree and shortest paths in the asynchronous model.

References

1 Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SICOMP*, 20(2):376–394, 1991.

2 Yehuda Afek and Yossi Matias. Elections in anonymous networks. *Information and Computation*, 113(2):312–330, 1994.

3 Dana Angluin. Local and global properties in networks of processors (extended abstract). In *STOC*, pages 82–93, 1980. doi:10.1145/800141.804655.

4 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

5 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 230–240, 1987.

6 Baruch Awerbuch. Distributed shortest paths algorithms (extended abstract). In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 490–500, 1989.

7 Baruch Awerbuch and Robert G Gallager. Distributed bfs algorithms. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 250–256. IEEE, 1985.

8 Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A trade-off between information and communication in broadcast protocols. *J. ACM*, 37(2):238–256, 1990.

9 Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 514–522. IEEE, 1990.

10 Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

11 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

12 Soumyottam Chatterjee, Gopal Pandurangan, and Peter Robinson. The complexity of leader election in diameter-two networks. *Distributed Computing*, pages 1–17, 2019.

13 F. Chin and H. F. Ting. Improving the time complexity of message-optimal distributed algorithms for minimum-weight spanning trees. *SIAM Journal on Computing*, 19(4):612–626, 1990.

14 Michael Elkin. A simple deterministic distributed mst algorithm, with near-optimal time and message complexities. In *PODC*, pages 157–163, 2017.

15 Michalis Faloutsos and Mart Molle. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing*, 17(2):151–170, 2004.

16 Eli Gafni. Improvements in the time complexity of two message-optimal election algorithms. In *Proceedings of the 4th Symposium on Principles of Distributed Computing (PODC)*, pages 175–185, 1985.

17 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Programming Languages & systems (TOPLAS)*, 5(1):66–77, January 1983. doi:10.1145/357195.357200.

18 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and broadcast with fewer messages, and faster gossiping. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC)*, pages 30:1–30:12, 2018.

19 Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

20 Seth Gilbert, Peter Robinson, and Suman Sourav. Leader election in well-connected graphs. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 227–236, 2018.

21 Robert Gmyr and Gopal Pandurangan. Time-message trade-offs in distributed algorithms. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 32:1–32:18, 2018.

22 Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round-and message-optimal distributed graph algorithms. In *PODC*, pages 119–128, 2018.

23 Pierre A. Humblet. Selecting a leader in a clique in $O(n \log n)$ messages. Memo, Lab. for Information & Decision Systems, MIT, 1984.

24 Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.

25 Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 263–272, New York, NY, USA, 2008. ACM. doi:10.1145/1400751.1400787.

26 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 71–80, 2015.

27 Ephraim Korach, Shay Kutten, and Shlomo Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Trans. Programming Languages & Systems (TOPLAS)*, 12(1):84–101, 1990. doi:10.1145/77606.77610.

28 Ephraim Korach, Shlomo Moran, and Shmuel Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 199–207, New York, NY, USA, 1984. ACM. doi:10.1145/800222.806747.

29 Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg. Singularly optimal randomized leader election. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.22.

30 Shay Kutten, William K. Moses Jr., Gopal Pandurangan, and David Peleg. Singularly near optimal leader election in asynchronous networks. *CoRR*, abs/2108.02197, 2021. arXiv:2108.02197.

31 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the complexity of universal leader election. *J. ACM*, 62:7, 2015.

32 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. Sublinear bounds for randomized leader election. *Theoretical Computer Science*, 561:134–143, 2015.

33 Gérard Le Lann. Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

34 Ivan Lavallée and Christian Lavault. Spanning tree construction for nameless networks. In *International Workshop on Distributed Algorithms*, pages 41–56. Springer, 1990.

35 Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., San Francisco, USA, 1996.

36 Ali Mashreghi and Valerie King. Time-communication trade-offs for minimum spanning tree construction. In *Proceedings of the 18th International Conference on Distributed Computing and Networking (ICDCN)*, 2017.

37 Ali Mashreghi and Valerie King. Brief announcement: Faster asynchronous MST and low diameter tree construction with sublinear communication. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 49:1–49:3, 2019.

38 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. *Distributed Computing*, pages 1–17, 2021.

39 Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

40 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 743–756, 2017.

41 David Peleg. Time-optimal leader election in general networks. *J. Parallel & Distributed Computing*, 8(1):96–99, 1990.

42 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

43 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 253–261. IEEE, 1999.

44 Gary Peterson. Efficient algorithms for elections in meshes and complete graphs. Technical report, TR 140, Dept. of CS, Univ. Rochester, 1985.

45 Murali Krishna Ramanathan, Ronaldo A. Ferreira, Suresh Jagannathan, Ananth Grama, and Wojciech Szpankowski. Randomized leader election. *Distributed Computing*, pages 403–418, 2007.

46 Nicola Santoro. *Design and Analysis of Distributed Algorithms (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2006.

47 Baruch Schieber and Marc Snir. Calling names on nameless networks. *Information and Computation*, 113(1):80–101, 1994.

48 Gurdip Singh. Efficient leader election using sense of direction. *Distributed Computing*, 10(3):159–165, 1997. doi:10.1007/s004460050033.

49 Gerard Tel. *Introduction to distributed algorithms.* Cambridge University Press, New York, NY, USA, 1994.

50 Donald M. Topkis. Performance analysis of information dissemination by flooding. *IEEE journal on selected areas in communications*, 7(3):335–340, 1989.

Permissionless and Asynchronous Asset Transfer

Petr Kuznetsov ✉
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Yvonne-Anne Pignolet ✉
DFINITY, Zürich, Switzerland

Pavel Ponomarev ✉
ITMO University, Saint Petersburg, Russia

Andrei Tonkikh ✉
National Research University Higher School of Economics, Saint Petersburg, Russia

Abstract

Most modern asset transfer systems use *consensus* to maintain a totally ordered chain of transactions. It was recently shown that consensus is not always necessary for implementing asset transfer. More efficient, *asynchronous* solutions can be built using *reliable broadcast* instead of consensus. This approach has been originally used in the closed (permissioned) setting. In this paper, we extend it to the open (*permissionless*) environment. We present PASTRO, a permissionless and asynchronous asset-transfer implementation, in which *quorum systems*, traditionally used in reliable broadcast, are replaced with a weighted *Proof-of-Stake* mechanism. PASTRO tolerates a *dynamic* adversary that is able to adaptively corrupt participants based on the assets owned by them.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Asset transfer, permissionless, asynchronous, dynamic adversary

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.28

Related Version This paper is an extended abstract of the technical report [19].
Technical Report: `https://arxiv.org/abs/2105.04966`

Funding *Petr Kuznetsov*: supported by TrustShare Innovation Chair.

1 Introduction

Inspired by advances in peer-to-peer data replication [25,31], a lot of efforts are currently invested in designing an algorithm for consistent and efficient exchange of assets in *dynamic* settings, where the set of participants, actively involved in processing transactions, varies over time.

Sometimes such systems are called *permissionless*, emphasizing the fact that they assume no trusted mechanism to regulate who and when can join the system. In particular, permissionless protocols should tolerate the notorious *Sybil attack* [7], where the adversary creates an unbounded number of "fake" identities.

Sharing data in a permissionless system is a hard problem. To solve it, we have to choose between consistency and efficiency. Assuming that the network is synchronous and that the adversary can only possess less than half of the total computing power, Bitcoin [25] and Ethereum [31] make sure that participants reach *consensus* on the order in which they access and modify the shared data. For this purpose, these systems employ a *proof-of-work* (PoW) mechanism to artificially slow down active participants (*miners*). The resulting algorithms are notoriously slow and waste tremendous amounts of energy.

Other protocols obviate the energy demands using *proof-of-stake* [2,4,18], *proof-of-space* [9], or *proof of space-time* [24]. However, these proposals still resort to synchronous networks, randomization, non-trivial cryptography and/or assume a trusted setup system.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 28; pp. 28:1–28:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we focus on a simpler problem, *asset transfer*, enabling a set of participants to exchange assets across their *accounts*. It has been shown [15, 16] that this problem does not require consensus. Assuming that every account is operated by a dedicated user, there is no need for the users to agree on a total order in which transactions must be processed: one can build an asset transfer system on top of the *reliable broadcast* abstraction instead of consensus. Unlike consensus [11], reliable broadcast allows for simple *asynchronous* solutions, enabling efficient asset transfer implementations that outperform their consensus-based counterparts [6].

Conventionally, a reliable broadcast algorithm assumes a *quorum system* [12, 22]. Every delivered message should be *certified* by a *quorum* of participants. Any two such quorums must have a correct participant in common and in every run, at least one quorum should consist of correct participants only. In a static f-resilient system of n participants, these assumptions result in the condition $f < n/3$. In a *permissionless* system, where the Sybil attack is enabled, assuming a traditional quorum system does not appear plausible. Indeed, the adversary may be able to control arbitrarily many identities, undermining any quorum assumptions.

In this paper, we describe a permissionless asset-transfer system based on *weighted* quorums. More precisely, we replace traditional quorums with certificates signed by participants holding a sufficient amount of assets, or *stake*. One can be alerted by this assumption, however: the notion of a "participant holding stake" at any given moment of time is not well defined in a decentralized consensus-free system where assets are dynamically exchanged and participants may not agree on the order in which transactions are executed. We resolve this issue using the notion of a *configuration*. A configuration is a partially ordered set of transactions that unambiguously determines the active system participants and the distribution of stake among them. As has been recently observed, configurations form a *lattice order* [20] and a *lattice agreement* protocol [10, 20, 21] can be employed to make sure that participants properly reconcile their diverging opinions on which configurations they are in.

Building on these abstractions, we present the PASTRO protocol to settle asset transfers, despite a *dynamic adversary* that can choose which participants to compromise *during* the execution, taking their current stake into account. The adversary is restricted, however, to corrupt participants that together own less than one third of stake in any *active candidate* configuration. Intuitively, a configuration (a set of transactions) is an active candidate configuration if all its transactions *can be* accepted by a correct process. At any moment of time, we may have multiple candidate configurations, and the one-third stake assumption must hold for each of them.

Note that a *superseded* configuration that has been successfully replaced with a new one can be compromised by the adversary. To make sure that superseded configurations cannot deceive slow participants that are left behind the reconfiguration process, we employ a *forward-secure* digital signature scheme [1, 8], recently proposed for Byzantine fault-tolerant reconfigurable systems [21]. The mechanism allows every process to maintain a single public key and a "one-directional" sequence of matching private keys: it is computationally easy to compute a new private key from an old one, but not vice versa. Intuitively, before installing a new configuration, one should ask holders of $> 2/3$ of stake of the old one to upgrade their private keys and destroy the old ones.

We believe that PASTRO is the right alternative to heavy-weight consensus-based replicated state machines, adjusted to applications that do not require global agreement on the order on their operations [20, 27]. Our solution does not employ PoW [25, 31] and does not rely on complex cryptographic constructions, such as a common coin [4,26]. More importantly, unlike recently proposed solutions [14,28], Pastro is resistant against a dynamic adversary that can choose which participants to corrupt in a dynamic manner, depending on the execution.

In this paper, we first present Pastro in its simplest version, as our primary message is a possibility result: a permissionless asset-transfer system can be implemented in an asynchronous way despite a dynamic adversary. We then discuss multiple ways of improving and generalizing our system. In particular, we address the issues of maintaining a dynamic amount of assets via an inflation mechanism and the performance of the system by delegation and incremental updates.

Road map. We overview related work in Section 2. In Section 3, we describe our model and recall basic definitions, followed by the formal statement of the asset transfer problem in Section 4. In Section 5, we describe Pastro and outline its correctness arguments in Section 6. We conclude with practical challenges to be addressed as well as related open questions in Section 7. Detailed proofs and the discussion of optimizations as well as delegation, fees, inflation, and practical aspects of using forward-secure digital signatures are deferred to the appendix.

2 Related Work

The first and still the most widely used permissionless cryptocurrencies are blockchain-based [5]. To make sure that honest participants agree (with high probability) on the order in which blocks of transactions are applied, the most prominent blockchains rely on proof-of-work [25,31] and assume synchronous communication. The approach exhibits bounded performance, wastes an enormous amount of energy, and its practical deployment turns out to be hardly decentralized (`https://bitcoinera.app/arewedecentralizedyet/`).

To mitigate these problems, more recent proposals suggest to rely on the *stake* rather than on energy consumption. E.g., in next version of Ethereum [30], random *committees* of a bounded number of *validators* are periodically elected, under the condition that they put enough funds at stake. Using nontrivial cryptographic protocols (verifiable random functions and common coins), Algorand [13] ensures that the probability for a user to be elected is proportional to its stake, and the committee is reelected after each action, to prevent adaptive corruption of the committee.

In this paper, we deliberately avoid reaching consensus in implementing asset transfer, and build atop asynchronous *reliable broadcast*, following [15,16]. It has been recently shown that this approach results in a simpler, more efficient and more robust implementation than consensus-based solutions [6]. However, in that design a static set of of processes is assumed, i.e., the protocol adheres to the *permissioned* model.

In [14], a reliable broadcast protocol is presented, that allows processes to join or leave the system without requiring consensus. ABC [28] proposes a direct implementation of a cryptocurrency, under the assumption that the adversary never corrupts processes holding 1/3 or more stake. However, both protocols [14,28] assume a static adversary: the set of corrupted parties is chosen at the beginning of the protocol.

In contrast, our solution tolerates an adversary that *dynamically* selects a set of corrupted parties, under the assumption that not too much stake is compromised in an *active candidate* configuration. Our solution is inspired by recent work on *reconfigurable* systems that base upon the reconfigurable lattice agreement abstraction [20,21]. However, in contrast to these *general-purpose* reconfigurable constructions, our implementation is much lighter. In our context, a configuration is just a distribution of stake, and every new transaction is a configuration update. As a result, we can build a simpler protocol that, unlike conventional asynchronous reconfigurable systems [20, 21, 29], does not bound the number of configuration updates for the sake of liveness.

3 Preliminaries

Processes and channels. We assume a set Π of potentially participating *processes*. In our model, every process acts both as a *replica* (maintains a local copy of the shared data) and as a *client* (invokes operations on the data).[1] In the proofs and definitions, we make the standard assumption of existence of a global clock not accessible to the processes.

At any moment of time, a process can be *correct* or *Byzantine*. We call a process *correct* as long as it faithfully follows the algorithm it has been assigned. A process is *forever-correct* if it remains correct forever. A correct process may turn *Byzantine*, which is modelled as an explicit event in our model (not visible to the other processes). A Byzantine process may perform steps not prescribed by its algorithm or prematurely stop taking steps. Once turned Byzantine, the process stays Byzantine forever.

We assume a *dynamic* adversary that can choose the processes to corrupt (to render Byzantine) depending on the current execution (modulo some restrictions that we discuss in the next section). In contrast, a *static* adversary picks up the set of Byzantine processes *a priori*, at the beginning of the execution.

In this paper we assume that the computational power of the adversary is bounded and, consequently, the cryptographic primitives used cannot be broken.

We also assume that each pair of processes is connected via *reliable authenticated channel*. If a forever-correct process p sends a message m to a forever-correct process q, then q eventually receives m. Moreover, if a correct process q receives a message m from a correct process p, then p has indeed sent m to q.

For the sake of simplicity, we assume that the set Π of potentially participating processes is finite.[2]

Weak reliable broadcast (WRB). In this paper we assume a *weak reliable broadcast primitive* to be available. The implementation of such primitive ensures the following properties:

- If a correct process delivers a message m from a correct process p, then m was previously broadcast by p;
- If a forever-correct process p broadcasts a message m, then p eventually delivers m;
- If a forever-correct process delivers m, then every forever-correct process eventually delivers m.

This weak reliable broadcast primitive can be implemented via a gossip protocol [17].

Forward-secure digital signatures. Originally, *forward-secure digital signature schemes* [1,8] were designed to resolve the key exposure problem: if the signature (private) key used in the scheme is compromised, then the adversary is capable to forge any previous (or future)

[1] We discuss how to split the processes into replicas and clients in the technical report [19].

[2] In practice, this assumptions boils down to requiring that the rate at which processes are added is not too high. Otherwise, if new stake-holders are introduced into the system at a speed prohibiting a client from reaching a sufficiently large fraction of them, we cannot make sure that the clients' transactions are eventually accepted.

signature. Using forward secure signatures, it is possible for the private key to be updated arbitrarily many times with the public key remaining fixed. Also, each signature is associated with a timestamp. This helps to identify messages which have been signed with the private keys that are already known to be compromised.

To generate a signature with timestamp t, the signer uses secret key sk_t. The signer can update its secret key and get sk_{t_2} from sk_{t_1} if $t_1 < t_2 \leq T$. However "downgrading" the key to a lower timestamp, from sk_{t_2} to sk_{t_1}, is computationally infeasible. As in recent work on Byzantine fault-tolerant reconfigurable systems [21], we model the interface of forward-secure signatures with an oracle which associates every process p with a timestamp st_p. The oracle provides the following functions:

- $\texttt{UpdateFSKey}(t)$ sets st_p to t if $t \geq st_p$;
- $\texttt{FSSign}(m, t)$ returns a signature for a message m and a timestamp t if $t \geq st_p$, otherwise $\perp$;
- $\texttt{FSVerify}(m, p, s, t)$ returns *true* if $s \neq \perp$ and it was generated by process p using $\texttt{FSSign}(m, t)$, *false* otherwise.

In most of the known implementations of forward-secure digital signature schemes the parameter T should be fixed in advance, which makes number of possible private key updates finite. At the same time, some forward-secure digital schemes [23] allow for an unbounded number of key updates ($T = +\infty$), however the time required for an update operation depends on the number of updates. Thus, local computational time may grow indefinitely albeit slowly. We discuss these two alternative implementations and reason about the most appropriate one for our problem in the technical report [19].

Verifiable objects. We often use *certificates* and *verifiable objects* in our protocol description. We say that object $obj \in O$ is *verifiable* in terms of a given *verifying function* $\texttt{Verify} : O \times \Sigma_O \to \{true, false\}$, if it comes together with a *certificate* $\sigma_{obj} \in \Sigma_O$, such that $\texttt{Verify}(obj, \sigma_{obj}) = true$, where Σ_O is a set of all possible certificates for objects of set O. A certificate σ_{obj} is *valid* for obj (in terms of a given verifying function) iff $\texttt{Verify}(obj, \sigma_{obj}) = true$. The actual meaning of an object "verifiability", as well as of a certificate validness, is determined by the verifying function.

4 Asset Transfer: Problem Statement

Before defining the asset transfer problem formally, we introduce the concepts used later.

Transactions. A *transaction* is a tuple $tx = (p, \tau, D)$. Here $p \in \Pi$ is an identifier of a process inside the system that initiates a transaction. We refer to p as the *owner* of tx. The map $\tau \colon \Pi \to \mathbb{Z}_0^+$ is the *transfer function*, specifying the amount of funds received by every process $q \in \Pi$ from this transaction.[3] D is a finite set of transactions that tx *depends on*, i.e., tx spends the funds p received through the transactions in D. We refer to D as the *dependency set* of tx.

Let $\mathcal{T}$ denote the set of all transactions. The function $value : \mathcal{T} \to V$ is defined as follows: $value(tx) = \sum_{q \in \Pi} tx.\tau(q)$. $\mathcal{T}$ contains one special "initial" transaction $tx_{init} = (\perp, \tau_{init}, \emptyset)$ with $value(tx_{init}) = M$ and no sender. This transaction determines the initial distribution of the total amount of "stake" in the system (denoted M).

[3] We encode this map as a set of tuples (q, d), where $q \in \Pi$ and $d > 0$ is the amount received sent to q in tx.

For all other transactions it holds that a transaction tx is *valid* if the amount of funds spent equals the amount of funds received: $\mathit{value}(tx) = \sum_{t \in t.D} t.\tau(tx.p)$. For simplicity, we assume that $\mathcal{T}$ contains only valid transactions: invalid transactions are ignored.

Transactions defined this way, naturally form a *directed graph* where each transaction is a node and edges represent dependencies.

We say that transactions tx_i and tx_j issued by the same process *conflict* iff the intersection of their dependency sets is non-empty: $tx_i \neq tx_j, tx_i.p = tx_j.p$ and $tx_i.D \cap tx_j.D \neq \emptyset$. A correct member of an asset-transfer system does not attempt to *double spend*, i.e., it never issues transactions which share dependencies and are therefore conflicting.

Transactions are equipped with a boolean function $\texttt{VerifySender} : \mathcal{T} \times \Sigma_{\mathcal{T}} \to \{\mathit{true}, \mathit{false}\}$. A transaction tx is *verifiable* iff it is verifiable in terms of the function $\texttt{VerifySender}(tx, \sigma_{tx})$. Here σ_{tx} is a certificate that confirms that tx was indeed issued by its owner process p. One may treat a transaction's certificate as p's digital signature of tx.

Asset-transfer system. An *asset-transfer* system (AS) maintains a partially ordered set of transactions T_p and exports one operation: $\texttt{Transfer}(tx, \sigma_{tx})$ adding transaction tx to the set T_p. Recall that $tx = (p, \tau, D)$, where p is the owner of transaction, τ is a transfer map, D is the set of dependencies, and σ_{tx} is a matching certificate.

In a distributed AS implementation, every process p holds the set of "confirmed" transactions T_p it is aware of, i.e., a local copy of the state. A transaction is said to be *confirmed* if some correct process p adds tx to its local copy of the state T_p. Set T_p can be viewed as the log of all confirmed transactions a process p is aware of. Let $T_p(t)$ denote the value of T_p at time t.

An AS implementation then satisfies the following properties:

Consistency: For every process p correct at time t, $T_p(t)$ contains only verifiable and non-conflicting transactions. Moreover, for every two processes p and q correct at time t and t' resp.: $(T_p(t) \subseteq T_q(t')) \vee (T_q(t') \subseteq T_p(t))$.

Monotonicity: For every correct process p, T_p can only grow with time: for all $t < t'$, $T_p(t) \subseteq T_p(t')$.

Validity: If a forever-correct process p invokes $\texttt{Transfer}(tx, \sigma_{tx})$ at time t, then there exists $t' > t$ such that $tx \in T_p(t')$.

Agreement: For a process p correct at time t and a forever-correct process q, there exists $t' \geq t$ such that $T_p(t) \subseteq T_q(t')$.

Here, $\sigma_{tx} \in \Sigma_{\mathcal{T}}$ is a certificate for transaction $tx = (p, \tau, D)$. A certificate protects the users from possible theft of their funds by other users. As we assume that cryptographic techniques (including digital signatures) are unbreakable, the only way to steal someone's funds is to steal their private key.

A natural convention is that a correct process never submits conflicting transactions. When it comes to Byzantine processes, we make no assumptions. Our specification ensures that if two conflicting transactions are issued by a Byzantine process, then at most one of them will ever be confirmed. In fact, just a single attempt of a process to cheat may lead to the loss of its funds as it can happen that neither of the conflicting transactions is confirmed and thus may preclude the process from making progress.

Transaction set as a configuration. For simplicity, we assume that the total amount of funds in the system is a publicly known constant $M \in \mathbb{Z}^+$ (fixed by the initial transaction tx_{init}) that we call *system stake*.[4]

[4] In the technical report [19], we discuss how to maintain a dynamic system stake.

A set of transactions $C \in 2^{\mathcal{T}}$ is called a *configuration*.

A configuration C is *valid* if no two transactions $tx_i, tx_j \in C$ are conflicting. In the rest of this paper, we only consider valid configurations, i.e., invalid configurations appearing in protocol messages are ignored by correct processes without being mentioned explicitly in the algorithm description.

The *initial configuration* is denoted by C_{init}, and consists of just one initial transaction ($C_{init} = \{tx_{init}\}$).

A valid configuration C determines the *stake* (also known as *balance*) of every process p as the difference between the amount of assets sent to p and the amount of assets sent by p in the transactions of C: $stake(q, C) = \sum_{tx \in C} tx.\tau(q) - \sum_{tx \in C \wedge tx.p = q} value(tx)$. Intuitively, a process joins the asset-transfer system as soon as it gains a positive stake in a configuration and leaves once its stake turns zero.

Functions *members* and *quorums* are defined as follows: $members(C) = \{p \mid \exists tx \in C$ such that $tx.\tau(p) > 0\}$, $quorums(C) = \{Q \mid \sum_{q \in Q} stake(q, C) > \frac{2}{3}M\}$. Intuitively, members of the system in a given configuration C are the processes that received money at least once, and a set of processes is considered to be a quorum in configuration C, if their total stake in C is more than two-thirds of the system stake.

Configuration lattice. Recall that a *join-semilattice* (we simply say a *lattice*) is a tuple $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set of *elements* provided with a partial-order relation $\sqsubseteq$, such that for any two elements $a \in \mathcal{L}$ and $b \in \mathcal{L}$, there exists the *least upper bound* for the set $\{a, b\}$, i.e., an element $c \in \mathcal{L}$ such that $a \sqsubseteq c$, $b \sqsubseteq c$ and $\forall d \in \mathcal{L}$: if $a \sqsubseteq d$ and $b \sqsubseteq d$, then $c \sqsubseteq d$. The least upper bound of elements $a \in \mathcal{L}$ and $b \in \mathcal{L}$ is denoted by $a \sqcup b$. $\sqcup$ is an associative, commutative and idempotent binary operator on $\mathcal{L}$. It is called the *join operator*.

The *configuration lattice* is then defined as $(\mathcal{C}, \sqsubseteq)$, where $\mathcal{C}$ is the set of all valid configurations, $\sqsubseteq = \subseteq$ and $\sqcup = \cup$.

5 Pastro Asset Transfer: Algorithm

In this section we present Pastro – an implementation of an asset-transfer system. We start with the main building blocks of the algorithm, and then proceed to the description of the Pastro protocol itself.

We bundle parts of Pastro algorithm that are semantically related in building blocks called *objects*, each offering a set of operations, and combine them to implement asset transfer.

Transaction Validation. The *Transaction Validation* (TV) object is a part of Pastro ensuring that transactions that a correct process p adds to its local state T_p do not conflict.

A correct process p submits a transaction $tx = (p, \tau, D)$ and a matching certificate to the object by invoking an operation `Validate`(tx, σ_{tx}). The operation returns a set of transactions $txs \in 2^{\mathcal{T}}$ together with a certificate $\sigma_{txs} \in \Sigma_{2^{\mathcal{T}}}$. We call a transaction set *verifiable* iff it is verifiable in terms of a function `VerifyTransactionSet`(txs, σ_{txs}). Intuitively, the function returns *true* iff certificate σ_{txs} confirms that set of transactions txs is *validated* by sufficiently many system members.

Formally, TV satisfies the following properties:

TV-Verifiability: If an invocation of `Validate` returns $\langle txs, \sigma_{txs} \rangle$ to a correct process, then `VerifyTransactionSet`$(txs, \sigma_{txs}) = true$;

TV-Inclusion: If `Validate`(tx, σ_{tx}) returns $\langle txs, \sigma_{txs} \rangle$ to a correct process, then $tx \in txs$;

TV-Validity: The union of returned verifiable transaction sets consists of non-conflicting verifiable transactions.

In our algorithm, we use one Transaction Validation object *TxVal*.

Adjustable Byzantine Lattice Agreement. Our asset-transfer algorithm reuses elements of an implementation of *Byzantine Lattice Agreement* (BLA), a Lattice Agreement [10] protocol that tolerates Byzantine failures. We introduce *Adjustable Byzantine Lattice Agreement* (ABLA), an abstraction that captures safety properties of BLA. An ABLA object is parameterized by a lattice $(\mathcal{L}, \sqsubseteq)$ and a boolean function $\texttt{VerifyInputValue} : \mathcal{L} \times \Sigma_{\mathcal{L}} \to \{\mathit{true}, \mathit{false}\}$. An input value of a given ABLA object is *verifiable* if it complies with `VerifyInputValue`.

ABLA exports one operation: $\texttt{Propose}(v, \sigma_v)$, where $v \in \mathcal{L}$ is an input value and $\sigma_v \in \Sigma_{\mathcal{L}}$ is a matching certificate. It also exports function $\texttt{VerifyOutputValue} : \mathcal{L} \times \Sigma_{\mathcal{L}} \to \{\mathit{true}, \mathit{false}\}$. The `Propose` operation returns a pair $\langle w, \sigma_w \rangle$, where $w \in \mathcal{L}$ is an output value and $\sigma_w \in \Sigma_{\mathcal{L}}$ is a matching certificate. An output value w of an ABLA object is *verifiable* if it complies with function `VerifyOutputValue`.

An ABLA object satisfies the following properties:

ABLA-Validity : Every verifiable output value w is a join of some set of verifiable input values;

ABLA-Verifiability: If an invocation of `Propose` returns $\langle w, \sigma_w \rangle$ to a correct process, then $\texttt{VerifyOutputValue}(w, \sigma_w) = \mathit{true}$;

ABLA-Inclusion: If $\texttt{Propose}(v, \sigma_v)$ returns $\langle w, \sigma_w \rangle$, then $v \sqsubseteq w$;

ABLA-Comparability: All verifiable output values are comparable.

In our algorithm, we use two ABLA objects, *ConfigLA* and *HistLA*.

Combining TV and ABLA. Let *ConfigLA* be an ABLA object parameterized as follows: $(\mathcal{L}, \sqsubseteq) = (2^{\mathcal{T}}, \subseteq)$, $\texttt{VerifyInputValue}(v, \sigma_v)$ is defined as $\texttt{VerifyTransactionSet}(v, \sigma_v)$, that is a part of TV. The function $\texttt{VerifyConfiguration}(C, \sigma_C)$ is an alias for $\mathit{ConfigLA}.\texttt{VerifyOutputValue}(C, \sigma_C)$. Using this object, the processes produce *comparable configurations*, i.e., transaction sets related by containment $(\subseteq)$.

HistLA is an ABLA object used to produce *sets of configurations* that are all related by containment. Under the assumption that every input (a set of configurations) to the object only contains comparable configurations, the outputs are related by containment and configurations in these sets are comparable. We can see these sets as *sequences* of ordered configurations. Such sets are called *histories*, as was recently suggested for asynchronous Byzantine fault-tolerant reconfiguration [21]. It was shown that histories allow us to access only $O(n)$ configurations, when n configurations are concurrently proposed. A history h is called *verifiable* if it complies with $\texttt{VerifyHistory}(h, \sigma_h)$.

Formally, the ABLA object *HistLA* is parameterized as follows: $(\mathcal{L}, \sqsubseteq) = (2^{2^{\mathcal{T}}}, \subseteq)$. The requirement that the elements of an input are comparable is established via the function $\texttt{VerifyInputValue}(\{\langle C_1, \sigma_1 \rangle, \dots, \langle C_n, \sigma_n \rangle\}) = \bigwedge_{i=1}^{n} \texttt{VerifyConfiguration}(C_i, \sigma_i)$. $\texttt{VerifyHistory}(H, \sigma_H)$ is an alias for $\mathit{HistLA}.\texttt{VerifyOutputValue}(H, \sigma_H)$.

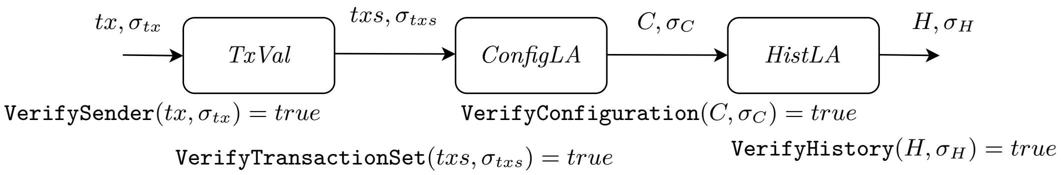

tx, σ_{tx}
TxVal
txs, σ_{txs}
ConfigLA
C, σ_C
HistLA
H, σ_H
$\texttt{VerifySender}(tx, \sigma_{tx}) = \mathit{true}$
$\texttt{VerifyConfiguration}(C, \sigma_C) = \mathit{true}$
$\texttt{VerifyTransactionSet}(txs, \sigma_{txs}) = \mathit{true}$
$\texttt{VerifyHistory}(H, \sigma_H) = \mathit{true}$

Figure 1 PASTRO pipeline.

Thus, the only valid input values for the *HistLA* object consist of verifiable output values of the *ConfigLA* object. At the same time, the only valid input values for the *ConfigLA* object are verifiable transaction sets, returned from the *TxVal* object. And, the only valid inputs for a *TxVal* object are signed transactions. In this case, all verifiable histories are related by containment (comparable via $\subseteq$), and all configurations within one verifiable history are related by containment (comparable via $\subseteq$) as well. Such a pipeline helps us to guarantee that all configurations of the system are comparable and contain non-conflicting transactions. Hence, such configurations can act as a consistent representation of stake distribution that changes with time.

To get a high-level idea of how Pastro works, imagine a conveyor belt (Figure 1). Transactions of different users are submitted as inputs, and as outputs, they obtain sets of configurations. Then one can choose the configuration representing the largest resulting set and install it in the system, changing the funds distribution in the system.

Algorithm overview. We assume that blocks of code (functions, operations, callbacks) are executed sequentially until they complete or get interrupted by a wait condition (**wait for** ...). Some events, e.g., receiving a message, may trigger callbacks (marked with keyword **upon**). However, they are not executed immediately, but firstly placed in an event queue, waiting for their turn.

By "**let** *var* = *expression*" we denote an assignment to a local variable (which can be accessed only from the current function, operation, or callback), and by "*var* ← *expression*" we denote an assignment to a global variable (which can be accessed from anywhere by the same process).

We denote calls to a weak reliable broadcast primitive with **WRB-broadcast**$\langle \dots \rangle$ and **WRB-deliver**$\langle \dots \rangle$. Besides, we assume a *weak uniform reliable broadcast (WURB)* primitive, a variant of uniform reliable broadcast [3]. WURB ensures an additional property compared to WRB: if the system remains static (i.e., the configurations stop changing) and a correct process delivers message m, then every forever-correct process eventually delivers m. The primitive helps us to ensure that if configuration C is never replaced with a greater one, then every forever-correct process will eventually learn about such a configuration C. To achieve this semantics, before triggering the **deliver** callback, a correct process just needs to ensure that a quorum of replicas have received the message [3]. The calls to WURB should be associated with some configuration C, and are denoted as **WURB-broadcast**$\langle \dots, C \rangle$ and **WURB-deliver**$\langle \dots, C \rangle$.

The algorithm uses one TV object *TxVal* and two ABLA objects: *ConfigLA* and *HistLA*. We list the pseudocode for the implementation of the *TxVal* object in Appendix A, Figure 5. The implementations of *ConfigLA* and *HistLA* are actually the same, and only differ in their parameters, we therefore provide one generalized implementation for both objects (Appendix A, Figure 6).

Every process maintains variables *history*, T_p and C_{cur}, accessible everywhere in the code. T_p is the latest installed configuration by the process p, by C_{cur} we denote the configuration starting from which the process needs to transfer data it stores to greater configurations, and *history* is the current verifiable history (along with its certificate σ_{hist}).

The main part of Pastro protocol (depicted in Figure 2) exports one operation `Transfer`(tx, σ_{tx}).

In this operation, we first set the local variables that store intermediate results produced in this operation, to null values $\perp$. Next, the `Validate` operation of the Transaction Validation object *TxVal* is invoked with the given transaction tx and the corresponding certificate σ_{tx}. The set of transactions txs and certificate σ_{txs} returned from the TV object are then used as an input for `Propose` operation of *ConfigLA*. Similarly, the result returned from *ConfigLA* “wrapped in” a singleton set, is used as an input for `Propose` operation of *HistLA*. The returned verifiable history is then broadcast in the system. We consider operation `Transfer`(tx, σ_{tx}) to *complete* by a correct process p once process p broadcasts message $\langle \textbf{NewHistory}, h, \sigma\rangle$ at line 9 or if p stops the current `Transfer` by executing line 17. Basically, the implementation of `Transfer` operation follows the logic described before and depicted in Figure 1.

```
typealias SignedTransaction = $\mathrm{Pair}\langle \mathcal{T}, \Sigma_{\mathcal{T}}\rangle$
typealias $\mathcal{C}$ = $\mathrm{Set}\langle \mathcal{T}\rangle$

Global variables:
$TxVal = \mathrm{TV}()$
$ConfigLA = \mathrm{ABLA}(\mathcal{L} = 2^{\mathcal{T}}, \sqsubseteq = \subseteq, v_{init} = C_{init})$
$ConfigLA.\mathrm{VerifyInputValue}(v, \sigma) = \mathrm{VerifyTransactionSet}(v, \sigma)$
$\mathrm{VerifyConfiguration}(v, \sigma) = ConfigLA.\mathrm{VerifyOutputValue}(v, \sigma)$
$HistLA = \mathrm{ABLA}(\mathcal{L} = 2^{2^{\mathcal{T}}}, \sqsubseteq = \subseteq, v_{init} = \{C_{init}\})$
$HistLA.\mathrm{VerifyInputValue}(\{v\}, \sigma) = \mathrm{VerifyConfiguration}(v, \sigma)$
$\mathrm{VerifyHistory}(v, \sigma) = HistLA.\mathrm{VerifyOutputValue}(v, \sigma)$

$T_p = C_{init}$
$C_{cur} = C_{init}$
$history = \{C_{init}\}$, $\sigma_{hist} = \perp$

$txs = \perp$, $\sigma_{txs} = \perp$
$C = \perp$, $\sigma_C = \perp$
$H = \perp$, $\sigma_H = \perp$
$curTx = \perp$

operation Transfer($tx$: $\mathcal{T}$, $\sigma_{tx}$: $\Sigma_{\mathcal{T}}$): void
1      $txs \leftarrow \perp$, $\sigma_{txs} \leftarrow \perp$
2      $C \leftarrow \perp$, $\sigma_C \leftarrow \perp$
3      $H \leftarrow \perp$, $\sigma_H \leftarrow \perp$
4      $curTx \leftarrow tx$
5      $txs, \sigma_{txs} \leftarrow TxVal.\mathrm{Validate}(tx, \sigma_{tx})$
6      $C, \sigma_C \leftarrow ConfigLA.\mathrm{Propose}(txs, \sigma_{txs})$
7      $H, \sigma_H \leftarrow HistLA.\mathrm{Propose}(\{C\}, \sigma_C)$
8      $curTx \leftarrow \perp$
9      WRB-broadcast $\langle \textbf{NewHistory}, H, \sigma_H\rangle$

upon WRB-deliver $\langle \textbf{NewHistory}, h, \sigma\rangle$ from any:
10     if VerifyHistory($h$, $\sigma$) and history $\subset h$ then
11         trigger event NewHistory(h) { $\forall C \in h : C$ is a candidate configuration }
12         history $\leftarrow h$, $\sigma_{hist} \leftarrow \sigma$
13         let $C_h$ = HighestConf(history)
14         UpdateFSKey(height($C_h$))
15         if $curTx \neq \perp$ then { There is an ongoing Transfer operation }
16             if $curTx \in C_h$ then { The last issued transaction by p is included in verifiable history }
17                 CompleteTransferOperation() { Stops the ongoing Transfer operation if any }
18             else if $txs = \perp$ then { Operation Transfer is ongoing and txs has not been received }
19                 $TxVal.\mathrm{Request}(\emptyset)$ { Restarts transaction validation by accessing $C_h$ }
20             else if $C = \perp$ then { Operation Transfer is ongoing and C has not been received yet }
21                 $ConfigLA.\mathrm{Refine}(\emptyset)$ { Restarts verifiable configuration reception by accessing $C_h$ }
22             else if $H = \perp$ then { Operation Transfer is ongoing and H has not been received yet }
23                 $HistLA.\mathrm{Refine}(\emptyset)$ { Restarts verifiable history reception by accessing $C_h$ }
```

Figure 2 PASTRO: code for process p.

If a correct process delivers a message $\langle \textbf{NewHistory}, h, \sigma\rangle$, where σ is a valid certificate for history h that is greater than its local estimate *history*, it “restarts” the first step that it has not yet completed in Transfer operation (lines 18-23). For example, if a correct process p receives a message $\langle \textbf{NewHistory}, h, \sigma_h\rangle$, where σ_h is a valid certificate for h and *history* $\subset h$

while being in *ConfigLA*, it restarts this step in order to access a greater configuration. The result of *ConfigLA* will still be returned to p in the place it has called *ConfigLA*.`Propose`. Intuitively, we do this in order to reach the most "up-to-date" configuration ("up-to-date" stake holders).

The State Transfer Protocol (Figure 3) helps us to ensure that the properties of the objects (*TxVal*, *ConfigLA* and *HistLA*) are satisfied *across configurations* that our system goes through. As system stake is redistributed actively with time, quorums in the system change as well and, hence, we need to pass the data that some quorum knows in configuration C to some quorum of any configuration $C' : C \sqsubset C'$, that might be installed after C. The protocol is executed by a correct process after it delivers a verifiable history h, such that $C \in h$ and $C_{cur} \sqsubset C$.

The *height* of a configuration C is the number of transactions in it (denoted as $\mathit{height}(C) = |C|$). Since all configurations installed in Pastro are comparable, height can be used as a unique identifier for an installed configuration. We also use height as the *timestamp* for forward-secure digital signatures. When a process p answers requests in configuration C, it signs the message with timestamp $\mathit{height}(C)$. The process p invokes UpdateFSKey$(\mathit{height}(C'))$ when it discovers a new configuration $C' : C \sqsubset C'$. Thus, processes that still consider C as the current configuration (and see the corresponding stake distribution) cannot be deceived by a process p, that was correct in C, but not in a higher installed configuration C' (e.g., p spent all its stake by submitting transactions, which became part of configuration C', thereby lost its weight in the system, and later became Byzantine).

The implementation of verifying functions (Figure 4) and a description of the auxiliary functions used in the pseudocode are delegated to Appendix A.

Implementing Transaction Validation. The implementation of the TV object *TxVal* is depicted in Figure 5 (Appendix A). The algorithm can be divided into two phases.

In the first phase, process p sends a message request that contains a set of transactions *sentTxs* to be validated to all members of the current configuration. Every correct process q that receives such messages first checks whether the transactions have actually been issued by their senders. If yes, q adds the transactions in the message to the set of transactions it has seen so far and checks whether any transaction from the ones it has just received conflicts with some other transaction it knows about. All conflicting transactions are placed in set *conflictTxs*. After q validates transactions, it sends a message $\langle \textbf{ValidateResp}, \mathit{txs}, \mathit{conflictTxs}, \mathit{sig}, \mathit{sn}\rangle$. Here, txs is the union of verifiable transactions received by p from q just now and all other non-conflicting transactions p is aware of so far. Process p then verifies a received message $\langle \textbf{ValidateResp}, \mathit{txs}_q, \mathit{conflictTxs}_q, \mathit{sig}_q, \mathit{sn}\rangle$. The message received from q is considered to be valid by p if q has signed it with a private key that corresponds to the current configuration, all the transactions from txs_q have valid certificates, and if for any verifiable transaction tx from *conflictTxs* there is a verifiable transaction tx also from *conflictTxs*, such that tx conflicts with tx'. If the received message is valid and txs_q equals *sentTxs*, then p adds signature of process q and its validation result to its local set acks_1. In case $\mathit{sentTxs} \subset \mathit{txs}_q$, the whole phase is restarted. The first phase is considered to be completed as soon as p collects responses from some quorum of processes in acks_1.

Such implementation of the first phase makes correct process p obtain certificate not only for its transaction, but also for other non-conflicting transactions issued by other processes. This helping mechanism ensures that transactions of forever-correct processes are eventually confirmed and become part of some verifiable configuration.

```
upon $C_{cur} \neq \text{HighestConf}(history)$:
    let $C_{next} = \text{HighestConf}(history)$
    let $S = \{C \in history \mid C_{cur} \sqsubseteq C \sqsubset C_{next}\}$
    $seqNum \leftarrow seqNum + 1$
    for $C$ in $S$:
        send $\langle$UpdateRead, $seqNum, C\rangle$ to $members(C)$
        wait for ($C \sqsubset C_{cur}$ or $\exists Q \in quorums(C)$ responded with $seqNum$)
    if $C_{cur} \sqsubset C_{next}$ then
        $C_{cur} \leftarrow C_{next}$
        WURB-broadcast $\langle$UpdateComplete, $C_{next}\rangle$

upon receive $\langle$UpdateRead, $sn, C\rangle$ from $q$:
    wait for $C \sqsubset \text{HighestConf}(history)$
    let $txs = TxVal.seenTxs$
    let $values_1 = ConfigLA.values$
    let $values_2 = HistLA.values$
    send $\langle$UpdateReadResp, $txs, values_1, values_2, sn\rangle$ to $q$

upon receive $\langle$UpdateReadResp, $txs, values_1, values_2, sn\rangle$ from $q$:
    if VerifySenders($txs$) and $ConfigLA$.VerifyValues($values_1$)
            and $HistLA$.VerifyValues($values_2$) then
        $TxVal.seenTxs \leftarrow TxVal.seenTxs \cup txs$
        $ConfigLA.values \leftarrow ConfigLA.values \cup values_1$.filter($\langle v, \sigma_v\rangle \Rightarrow v \notin ConfigLA.values$.firsts())
        $HistLA.values \leftarrow HistLA.values \cup values_2$.filter($\langle v, \sigma_v\rangle \Rightarrow v \notin HistLA.values$.firsts())

upon WURB-deliver $\langle$UpdateComplete, $C\rangle$ from quorum $Q \in quorums(C)$:
    wait for $C \in history$
    if $T_p \sqsubset C$ then
        if $C_{cur} \sqsubset C$ then $C_{cur} \leftarrow C$
        $T_p \leftarrow C$ { Update set of "confirmed" transactions }
        trigger event InstalledConfig($C$) { $C$ is an installed configuration }
```

Figure 3 State Transfer Protocol: code for process p.

In the second phase, p collects signatures from a weighted quorum of the current configuration. If p successfully collects such a set of signatures, then the configuration it saw during the first phase was *active* (no greater configuration had been installed) and it is safe for p to return the obtained result. This way the so-called "slow reader" attack [21] is anticipated.

If during any of the two phases p receives a message with a new verifiable history that is greater (w.r.t. $\subseteq$) than its local estimate and does not contain last issued transaction by p, the described algorithm starts over. We guarantee that the number of restarts in *TxVal* in Pastro protocol is finite only for *forever-correct* processes (please refer to the technical report [19] for a detailed proof). Note that we cannot guarantee this for *all* correct processes as during the protocol execution some of them can become Byzantine.

In the implementation, we assume that the dependency set of a transaction only includes transactions that are confirmed (i.e., included in some installed configuration), otherwise they are considered invalid.

Implementing Adjustable Byzantine Lattice Agreement. The generalized implementation of ABLA objects *ConfigLA* and *HistLA* is specified by Figure 6 (Appendix A). The algorithm is generally inspired by the implementation of Dynamic Byzantine Lattice Agreement from [21], but there are a few major differences. Most importantly, it is tailored to work even if the number of reconfiguration requests (i.e., transactions) is infinite. Similarly to the Transaction Validation implementation, algorithm consists of two phases.

In the first phase, process p sends a message that contains the verifiable inputs it knows to other members and then waits for a weighted quorum of processes of the current configuration to respond with the same set. If p receives a message with a greater set (w.r.t. $\subseteq$), it restarts the phase. The validation process performed by the processes is very similar to the one used in Transaction Validation object implementation.

The second phase, in fact, is identical to the second phase of the Transaction Validation implementation. We describe it in the pseudocode for completeness.

As with Transaction Validation, whenever p delivers a verifiable history h, such that it is greater than its own local estimate and h does not contain last issued transaction by p, the described algorithm starts over. Similarly to *TxVal*, it is guaranteed that the number of restarts a forever-correct process make in both *ConfigLA* and *HistLA* is finite.

6 Proof of Correctness

In this section, we outline the proof that PASTRO indeed satisfies the properties of an asset-transfer system. First, we formulate a restriction we impose on the adversary that is required for our implementation to be correct. Informally, the adversary is not allowed to corrupt one third or more stake in a "candidate" configuration, i.e., in a configuration that can potentially be used for adding new transactions. The adversary is free to corrupt a configuration as soon as it is superseded by a strictly higher one. We then sketch the main arguments of our correctness proof (the detailed proof is available in [19]).

6.1 Adversarial restrictions

A configuration C is considered to be *installed* if some correct process has triggered the special event InstalledConfig(C). We call a configuration C a *candidate* configuration if some correct process has triggered a NewHistory(h) event, such that $C \in h$. We also say that a configuration C is *superseded* if some correct process installed a higher configuration C'. An installed (resp., candidate) configuration C is called an *active* (resp., an *active candidate*) configuration as long as it is not superseded. Note that at any moment of time t, every active installed configuration is an active candidate configuration, but not vice versa.

We expect the adversary to obey the following condition:

Configuration availability: Let C be an active candidate configuration at time t and let $\mathit{correct}(C, t)$ denote the set of processes in $\mathit{members}(C)$ that are correct at time t. Then C must be *available* at time t:

$$\sum_{q \in \mathit{correct}(C,t)} \mathit{stake}(q, C) > 2/3M.$$

Note that the condition allows the adversary to compromise a candidate configuration once it is superseded by a more recent one. As we shall see, the condition implies that our algorithm is live. Intuitively, a process with a pending operation will either eventually hear from the members of a configuration holding "enough" stake which might allow it to complete its operation or will learn about a more recent configuration, in which case it can abandon the superseded configuration and proceed to the new one.

6.2 Proof outline

Consistency. The consistency property states that (1) as long as process p is correct, T_p contains only verifiable non-conflicting transactions and (2) if processes p and q are correct at times t and t' respectively, then $T_p(t) \subseteq T_q(t')$ or $T_q(t') \subseteq T_p(t)$. To prove that PASTRO satisfies this property, we show that our implementation of *TxVal* meets the specification of Transaction Validation and that both *ConfigLA* and *HistLA* objects satisfy the properties of Adjustable Byzantine Lattice Agreement. Correctness of *HistLA* ensures that all verifiable histories are related by containment and the correctness of *ConfigLA* guarantees that all verifiable configurations are related by containment (i.e., they are *comparable*). Taking into account that the only possible verifiable inputs for *HistLA* are sets that contain verifiable output values (configurations) of *ConfigLA*, we obtain the fact that all configurations of any verifiable history are comparable as well. As all installed configurations (all C such that a correct process triggers an event InstalledConfig(C)) are elements of some verifiable history, they all are related by containment too. Since T_p is in fact the last configuration installed by a process p, we obtain (2). The fact that every p stores verifiable non-conflicting transactions follows from the fact that the only possible verifiable input values for *ConfigLA* are the output transaction sets returned by *TxVal*. As *TxVal* is a correct implementation of Transaction Validation, then union of all such sets contain verifiable non-conflicting transactions. Hence, the only verifiable configurations that are produced by the algorithm cannot contain conflicting transactions. From this we obtain (1).

Monotonicity. This property requires that T_p only grows as long as p is correct. The monotonicity of Pastro follows from the fact that correct processes install only greater configurations with time, and that the last installed configuration by a correct process p is exactly T_p. Thus, if p is correct at time t', then for all $t < t'$: $T_p(t) \subseteq T_p(t')$.

Validity. This property requires that a transfer operation for a transaction tx initiated by a forever-correct process will lead to tx being included in T_p at some point in time. In order to prove this property for Pastro, we show that a forever-correct process p may only be blocked in the execution of a `Transfer` operation if some other process successfully installed a new configuration. We argue that from some moment of time on every other process that succeeds in the installation of configuration C will include the transaction issued by p in the C. We also show that if such a configuration C is installed then eventually every forever-correct process installs a configuration $C' : C \sqsubseteq C'$. As T_p is exactly the last configuration installed by p, eventually any transaction issued by a forever-correct process p is included in T_p.

Agreement. In the end we show that the Pastro protocol satisfies the agreement property of an asset-transfer system. The property states the following: for a correct process p at time t and a forever-correct process q, there exists $t' \geq t$ such that $T_p(t) \subseteq T_p(t')$. Basically, it guarantees that if a transaction tx was considered confirmed by p when it was correct, then any forever-correct process will eventually confirm it as well. To prove this, we show that if a configuration C is installed by a correct process, then every other forever-correct process will install some configuration C', such that $C \sqsubseteq C'$. Taking into account the fact that T_p is a last configuration installed by a process p, we obtain the desired.

7 Concluding Remarks

Pastro is a permissionless asset transfer system based on proof of stake. It builds on lattice agreement primitives and provides its guarantees in asynchronous environments where less than one third of the total stake is owned by malicious parties.

Enhancements and optimizations. To keep the presentation focused, so far we described only the core of the Pastro protocol. However, there is a number of challenges that need to be addressed before the protocol can be applied in practice. In particular the communication, computation and storage complexity of the protocol can be improved with carefully constructed messages and signature schemes. Also, mechanisms for stake delegation as well as fees and inflation are necessary. Note that these are non-trivial to implement in an asynchronous system because there is no clear agreement on the order in which transactions are added to the system or on the distribution of stake at the moment when a transaction is added to the system. We discuss these topics as well as the practical aspects of using forward-secure signatures in the technical report [19].

Open questions. In this paper, we demonstrated that it is possible to combine asynchronous cryptocurrencies with proof-of-stake in presence of a dynamic adversary. However, there are still plenty of open questions. Perhaps, the most important direction is to study hybrid solutions which combine our approach with consensus in an efficient way in order to support general-purpose smart contracts. Further research is also needed in order to improve the efficiency of the solution and to measure how well it will behave in practice compared to consensus-based solutions. Finally, designing proper mechanisms in order to incentivize active and honest participation is a non-trivial problem in the harsh world of asynchrony, where the processes cannot agree on a total order in which transactions are executed.

References

1 Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448, Berlin, 1999. Springer.

2 Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography and Data Security – FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 142–157, Berlin, 2016. Springer.

3 Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

4 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.

5 CoinMarketCap. Cryptocurrency prices, charts and market capitalizations, 2021. , accessed 2021-02-15. URL: `https://coinmarketcap.com/`.

6 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 – July 2, 2020*, pages 26–38. IEEE, 2020.

7 John R. Douceur. The sybil attack. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, pages 251–260, Heidelberg, 2002. Springer.

8 Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers`.

9 Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology – CRYPTO 2015 – 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 585–605, 2015.

10 Jose M. Falerio, Sriram K. Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 125–134. ACM, 2012.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

12 David K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162, 1979.

13 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

14 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic Byzantine reliable broadcast. In *OPODIS*, 2020.

15 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC*, 2019. `arXiv:1906.05574`.

16 Saurabh Gupta. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing. Master's thesis, Arizona State University, USA, 2016.

17 Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, 2007. `doi:10.1145/1317379.1317381`.

18 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017 – 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

19 Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer [technical report]. *arXiv preprint*, 2021. `arXiv:2105.04966`.

20 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Reconfigurable lattice agreement and applications. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 31:1–31:17, 2019.

21 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 27:1–27:17, 2020.

22 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11?(4):203–213, 1998.

23 Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Advances in Cryptology – Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 400–417, Amsterdam, The Netherlands, April 28 – May 2 2002. IACR, Springer-Verlag.

24 Tal Moran and Ilan Orlov. Proofs of space-time and rational proofs of storage. *IACR Cryptology ePrint Archive*, 2016:35, 2016.

25 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

26 Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

27 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

28 Jakub Sliwinski and Roger Wattenhofer. ABC: asynchronous blockchain without consensus. *CoRR*, abs/1909.10926, 2019. `arXiv:1909.10926`.

29 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.

30 Paul Wackerow, Ryan Cordell, Tentodev, Alwin Stockinger, and Sam Richards. Ethereum proof-of-stake (pos), 2008. accessed 2021-02-15. URL: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`.

31 Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.

A Pseudocode

Verifying and auxiliary functions. We provide implementations of required verifying functions used in *ConfigLA* and *HistLA* in Figure 4. The implementation of function `VerifySender`(tx, σ_{tx}) is not presented there, as it simply consists in verifying that σ_{tx} is a valid signature for tx under $tx.p$'s public key.

We use the following auxiliary functions in the pseudocode to keep it concise:

- `VerifySenders`(txs) – returns *true* iff $\forall\langle tx, \sigma_{tx}\rangle \in txs$: `VerifySender`$(tx, \sigma_{tx}) = true$, otherwise returns *false*;
- `VerifyValues`(vs) – returns *true* if $\forall\langle v, \sigma\rangle \in vs$: `VerifyInputValue`$(v, \sigma) = true$, otherwise returns *false*;
- `ContainsQuorum`$(acks, C)$ – returns *true* if $\exists Q \in quorums(C)$ such that $\forall q \in Q\langle q, \dots\rangle \in acks$, otherwise returns *false*;
- `HighestConf`(h) – returns the highest (w.r.t. $\sqsubseteq$) configuration in given history h;
- `firsts`() – method that, when invoked on a set of tuples S, returns a set of first elements of tuples from set S;
- `ConflictTransactions`(txs) – for a given set of verifiable transactions txs returns a set $conflictTxs$ such that $conflictTxs \subseteq txs$ and for any $\langle tx, \sigma_{tx}\rangle \in conflictTxs$ there exists $\langle tx', \sigma_{tx'}\rangle \in conflictTxs$ such that tx conflicts with tx';
- `CorrectTransactions`(txs) – returns a set of transactions $correctTxs$ such that $correctTxs \subseteq txs$, $\langle tx, \sigma_{tx}\rangle \in correctTxs$ iff $\langle tx, \sigma_{tx}\rangle \in txs$, σ_{tx} is a valid certificate for tx and $\nexists\langle tx', \sigma_{tx'}\rangle \in txs$, such that tx conflicts with tx'.

```
fun VerifyTransactionSet($txs : \mathrm{Set}\langle\mathcal{T}\rangle, \sigma_{txs} : \Sigma_{2^\mathcal{T}}$) : Bool
    let $\langle sentTxs, acks_1, acks_2, h, \sigma_h\rangle = \sigma_{txs}$
    let $C = \mathrm{HighestConf}(h)$
    return $\mathrm{VerifyHistory}(h, \sigma_h)$ and $txs = sentTxs.\mathrm{firsts}() \setminus acks_1.\mathrm{getConflictTxs}().\mathrm{firsts}()$
        and $\mathrm{ContainsQuorum}(acks_1, C)$ and $\mathrm{ContainsQuorum}(acks_2, C)$
        and $acks_1.\mathrm{forAll}(\langle q, sig, conflictTxs\rangle \Rightarrow$
            $\mathrm{FSVerify}(\langle \textbf{ValidateResp}, sentTxs, conflictTxs\rangle, q, sig, height(C)))$
        and $acks_1.\mathrm{forAll}(\langle q, sig, conflictTxs\rangle \Rightarrow$
            $conflictTxs.\mathrm{forAll}(\langle tx, \sigma_{tx}\rangle \Rightarrow tx$ conflicts with $tx'$ such that $\langle tx', \sigma_{tx'}\rangle \in conflictTxs))$
        and $acks_2.\mathrm{forAll}(\langle q, sig\rangle \Rightarrow \mathrm{FSVerify}(\langle \textbf{ConfirmResp}, acks_1\rangle, q, sig, height(C)))$

fun ABLA.VerifyOutputValue($v : \mathcal{L}, \sigma : \Sigma$) : Bool
    if $\sigma = \perp$ then return $v = v_{init}$
    let $\langle values, acks_1, acks_2, h, \sigma_h\rangle = \sigma$
    let $C = \mathrm{HighestConf}(h)$
    return $\mathrm{VerifyHistory}(h, \sigma_h)$ and $v = \bigsqcup values.\mathrm{firsts}()$
        and $\mathrm{ContainsQuorum}(acks_1, C)$ and $\mathrm{ContainsQuorum}(acks_2, C)$
        and $acks_1.\mathrm{forAll}(\langle q, sig\rangle \Rightarrow \mathrm{FSVerify}(\langle \textbf{ProposeResp}, values\rangle, q, sig, height(C)))$
        and $acks_2.\mathrm{forAll}(\langle q, sig\rangle \Rightarrow \mathrm{FSVerify}(\langle \textbf{ConfirmResp}, acks_1\rangle, q, sig, height(C)))$

fun VerifyConfiguration($v, \sigma$) = $ConfigLA.\mathrm{VerifyOutputValue}(v, \sigma)$

fun VerifyHistory($v, \sigma$) = $HistLA.\mathrm{VerifyOutputValue}(v, \sigma)$
```

Figure 4 Verifying functions: code for process p.

```
**Global variables**:
$seenTxs, correctTxs, sentTxs : \text{Set}\langle\text{SignedTransaction}\rangle = \{\langle tx_{init}, \bot\rangle\}$
$acks_1 = \emptyset,\ acks_2 = \emptyset,\ status = inactive$

**operation** Validate($tx : \mathcal{T}, \sigma : \Sigma_{\mathcal{T}}$): Pair$\langle \text{Set}\langle\mathcal{T}\rangle, \Sigma_{2\mathcal{T}}\rangle$
    Request($\{\langle tr, \sigma\rangle\}$)
    **wait for** ContainsQuorum(acks$_2$, HighestConf($history$))
    $status \leftarrow inactive$
    **let** $\sigma = \langle sentTxs, acks_1, acks_2, history, \sigma_{hist}\rangle$
    **return** $\langle sentTxs.\text{firsts}()\ \setminus\ acks_1.\text{getConflictTxs}().\text{firsts}(),\ \sigma\rangle$

**fun** Request($txs$: Set$\langle$SignedTransaction$\rangle$): void
    $seenTxs \leftarrow seenTxs \cup txs,\ correctTxs \leftarrow \text{CorrectTransactions}(seenTxs)$
    $sentTxs \leftarrow correctTxs,\ acks_1 \leftarrow \emptyset,\ acks_2 \leftarrow \emptyset$
    $seqNum \leftarrow seqNum + 1,\ status \leftarrow requesting$
    **let** $C = \text{HighestConf}(history)$
    **send** $\langle$**ValidateReq**$, sentTxs,\ seqNum, C\rangle$ to members($C$)

**upon** ContainsQuorum(acks$_1$, HighestConf($history$)):
    status $\leftarrow$ $confirming$, **let** $C = \text{HighestConf}(history)$
    **send** $\langle$**ConfirmReq**$, acks_1, seqNum, C\rangle$ to members($C$)

**upon** receive $\langle$**ValidateReq**$, txs, sn, C\rangle$ from $q$:
    **wait for** $C = T_p$ **or** HighestConf($history$) $\not\sqsubseteq C$
    **if** VerifySenders($txs$) **then**
        $seenTxs \leftarrow seenTxs \cup txs,\ correctTxs \leftarrow \text{CorrectTransactions}(seenTxs)$
        **if** $C = \text{HighestConf}(history)$ **then**
            **let** $conflictTxs = \text{ConflictTransactions}(seenTxs)$
            **let** $sig = \text{FSSign}(\langle$**ValidateResp**$,\ txs \cup correctTxs,\ conflictTxs\rangle, \text{height}(C))$
            **send** $\langle$**ValidateResp**$,\ txs \cup correctTxs,\ conflictTxs,\ sig,\ sn\rangle$ to $\boldsymbol{q}$

**upon** receive $\langle$**ValidateResp**$, txs, conflictTxs, sig, sn\rangle$ from $q$ **and** $sn = seqNum$ **and** $status = requesting$:
    **let** $C = \text{HighestConf}(history)$
    **let** $isValid = \text{FSVerify}(\langle$**ValidateResp**$, txs,\ conflictTxs\rangle, q, sig, \text{height(C)})$
            **and** VerifySenders($txs \cup conflictTxs$) **and** $conflictTxs.\text{forAll}(\langle tx, \sigma_{tx}\rangle \Rightarrow$
                $tx$ conflicts with some $tx'$ such that $\langle tx', \sigma_{tx'}\rangle \in conflictTxs)$
    **if** $isValid$ **and** $txs = sentTxs$ **then** $acks_1 \leftarrow acks_1 \cup \{\langle q, sig, conflictTxs\rangle\}$
    **else if** $isValid$ **and** $sentTxs \subsetneq txs$ **then** Request($txs$)

**upon** receive $\langle$**ConfirmReq**$,\ acks,\ sn,\ C\rangle$ from $q$:
    **wait for** $C = T_p$ **or** HighestConf($history$) $\not\sqsubseteq C$
    **if** $C = \text{HighestConf}(history)$ **then**
        **let** $sig = \text{FSSign}(\langle$**ConfirmResp**$,\ acks\rangle, \text{height}(C))$
        **send** $\langle$**ConfirmResp**$,\ sig,\ sn\rangle$ to $q$

**upon** receive $\langle$**ConfirmResp**$,\ sig,\ sn\rangle$ from $q$ **and** $sn = seqNum$ **and** $status = confirming$:
    **let** $C = \text{HighestConf}(history)$
    **let** $isValidSig = \text{FSVerify}(\langle$**ConfirmResp**$, acks_1\rangle,\ q,\ sig, \text{height}(C))$
    **if** $isValidSig$ **then** $acks_2 \leftarrow acks_2 \cup \{\langle q,\ sig\rangle\}$
```

Figure 5 Transaction Validation: *TxVal* implementation.

```
**Parameters**:
Lattice $(\mathcal{L}, \sqsubseteq)$ and initial value $v_{init} \in \mathcal{L}$
Function VerifyInputValue$(v, \sigma_v)$
**Global variables**:
$values : \text{Set}\langle\text{Pair}\langle\mathcal{L}, \Sigma\rangle\rangle = \{\langle v_{init}, \perp\rangle\}$
$acks_1 = \emptyset$, $acks_2 = \emptyset$, $status = inactive$

operation Propose$(v : \mathcal{L}, \sigma : \Sigma)$: Pair$\langle\mathcal{L}, \Sigma\rangle$
    Refine$(\{\langle v, \sigma\rangle\})$
    wait for ContainsQuorum$(acks_2$, HighestConf$(history))$
    $status \leftarrow inactive$
    let $\sigma = \langle values, acks_1, acks_2, history, \sigma_{hist}\rangle$
    return $\langle(\bigsqcup values.\text{firsts}()), \sigma\rangle$

fun Refine$(vs : \text{Set}\langle\text{Pair}\langle\mathcal{L}, \Sigma\rangle\rangle)$: void
    $acks_1 \leftarrow \emptyset$, $acks_2 \leftarrow \emptyset$
    $values \leftarrow values \cup vs.\text{filter}(\langle v, \sigma_v\rangle \Rightarrow v \notin values.\text{firsts}())$
    $seqNum \leftarrow seqNum + 1$, $status \leftarrow proposing$
    let $C$ = HighestConf$(history)$
    send ⟨ProposeReq, $values$, $seqNum, C$⟩ to members$(C)$

upon ContainsQuorum$(acks_1$, HighestConf$(history))$
    status $\leftarrow$ $confirming$, let $C$ = HighestConf$(history)$
    send ⟨ConfirmReq, $acks_1$, $seqNum, C$⟩ to members$(C)$

upon receive ⟨ProposeReq, $vs$, $sn$, $C$⟩ from $q$:
    wait for $C = T_p$ or HighestConf$(history) \not\sqsubseteq C$
    if VerifyValues$(vs \setminus values)$ then
        $values \leftarrow values \cup vs.\text{filter}(\langle v, \sigma_v\rangle \Rightarrow v \notin values.\text{firsts}())$
        let $replyValues = vs \cup values.\text{filter}(\langle v, \sigma_v\rangle \Rightarrow v \notin vs.\text{firsts}())$
        if $C$ = HighestConf$(history)$ then
            let $sig$ = FSSign(⟨ProposeResp, $replyValues$⟩, height$(C)$)
            send ⟨ProposeResp, $replyValues$, $sig$, $sn$⟩ to $q$

upon receive ⟨ProposeResp, $vs$, $sig$, $sn$⟩ from $q$ and $sn = seqNum$ and $status = proposing$:
    let $C$ = HighestConf$(history)$
    let $isValidSig$ = FSVerify(⟨ProposeResp, $vs$⟩, $q$, $sig$, height$(C)$)
    if $isValidSig$ then
        if $values = vs$ then $acks_1 \leftarrow acks_1 \cup \{\langle q, sig\rangle\}$
        else if VerifyValues$(vs \setminus values)$ then Refine$(vs \setminus values)$

upon receive ⟨ConfirmReq, $acks$, $sn$, $C$⟩ from $q$:
    wait for $C = T_p$ or HighestConf$(history) \not\sqsubseteq C$
    if $C$ = HighestConf$(history)$ then
        let $sig$ = FSSign(⟨ConfirmResp, $acks$⟩, height$(C)$)
        send ⟨ConfirmResp, $sig$, $sn$⟩ to $q$

upon receive ⟨ConfirmResp, $sig$, $sn$⟩ from $q$ and $sn = seqNum$ and $status = confirming$:
    let $C$ = HighestConf$(history)$
    let $isValidSig$ = FSVerify(⟨ConfirmResp, $acks_1$⟩, $q$, $sig$, height$(C)$))
    if $isValidSig$ then $acks_2 \leftarrow acks_2 \cup \{\langle q, sig\rangle\}$
```

Figure 6 Adjustable Byzantine Lattice Agreement: generalized implementation of *ConfigLA* and *HistLA*.

Detectable Sequential Specifications for Recoverable Shared Objects

Nan Li
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Abstract

The recent commercial release of persistent main memory by Intel has sparked intense interest in recoverable concurrent objects. Such objects maintain state in persistent memory, and can be recovered directly following a system-wide crash failure, as opposed to being painstakingly rebuilt using recovery state saved in slower secondary storage. Specifying and implementing recoverable objects is technically challenging on current generation hardware precisely because the top layers of the memory hierarchy (CPU registers and cache) remain volatile, which causes application threads to lose critical execution state during a failure. For example, a thread that completes an operation on a shared object and then crashes may have difficulty determining whether this operation took effect, and if so, what response it returned. Friedman, Herlihy, Marathe, and Petrank (DISC'17) recently proposed that this difficulty can be alleviated by making the recoverable objects *detectable*, meaning that during recovery, they can resolve the status of an operation that was interrupted by a failure. In this paper, we formalize this important concept using a *detectable sequential specification (DSS)*, which augments an object's interface with auxiliary methods that threads use to first declare their need for detectability, and then perform detection if needed after a failure. Our contribution is closely related to the nesting-safe recoverable linearizability (NRL) framework of Attiya, Ben-Baruch, and Hendler (PODC'18), which follows an orthogonal approach based on ordinary sequential specifications combined with a novel correctness condition. Compared to NRL, our DSS-based approach is more portable across different models of distributed computation, compatible with several existing linearizability-like correctness conditions, less reliant on assumptions regarding the system, and more flexible in the sense that it allows applications to request detectability on demand. On the other hand, application code assumes full responsibility for nesting DSS-based objects. As a proof of concept, we demonstrate the DSS in action by presenting a detectable recoverable lock-free queue algorithm and evaluating its performance on a multiprocessor equipped with Intel Optane persistent memory.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Computer systems organization → Reliability

Keywords and phrases persistent memory, concurrency, fault tolerance, correctness, detectability

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.29

Funding *Wojciech Golab*: Author supported by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

1 Introduction

The past several years have witnessed an eruption of research into software techniques for harnessing the power of persistent memory, a byte-addressable medium that combines the performance of conventional DRAM with the data durability of secondary storage devices. Prior to the emergence of persistent memory, traditional memory hierarchies forced applications to maintain state redundantly using both in-memory structures for performance and on-disk structures for durability, which imposes a performance overhead during failure-

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 29; pp. 29:1–29:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

free operation, and also slows down recovery after a failure. The convergence of primary and secondary storage in persistent memory creates a new avenue for eliminating the inefficiency of rebuilding in-memory state from zero after each system-wide failure, and calls for thoughtful reconsideration of established software design principles.

Harnessing the performance of persistent memory while maintaining correctness to preserve application data integrity poses a number of technical challenges. To begin with, well-studied legacy techniques based on database-style recovery logging provide insufficient parallelism when applied to high-performance concurrent data structures. Novel techniques are needed to remove this bottleneck, and these often intertwine tightly with concurrency control and memory management mechanisms. The implementation task is further complicated on current generation hardware by volatile caching, which necessitates the use of explicit persistence instructions to flush updates to the persistent medium, and by the lack of support in the processor's instruction set for multi-word failure-atomic writes [13, 41]. Hardware transactional memory (HTM) [28] does not solve the latter problem on current generation multiprocessors because flushing instructions abort transactions before they can be committed.

Rising eagerly to the new challenge, researchers have devised a variety of techniques and idioms for both implementing persistent data structures and specifying their correct behavior. Early practical contributions in this space include a variety of low-level durability mechanisms and APIs for building fault-tolerant applications [10, 12, 29, 40, 44]. This work eventually spurred a wave of theoretical research on concurrent objects for persistent memory, starting with different perspectives on formalizing the behavior of such objects under concurrent access, and continuing with provably correct implementation techniques [4, 5, 6, 7, 8, 11, 20, 23, 30]. One of the fundamental scientific questions arising from this work is how to imbue persistent data structures with *detectability* – the ability to resolve the outcome of operations that were interrupted by a failure [19, 20]. This type of forensic capability is especially important in systems that lack transactions, because the application is directly responsible for deciding the correct redo and undo actions.

Friedman et al.'s practitioner-oriented definition of detectability [20] conveys clearly the high-level intention, and also explains some of the implementation details, in a model of computation where crash failures are system-wide and recovery actions occur during a *recovery phase* that is initiated by the system and precedes resumption of ordinary activity by application threads. Specifically, threads announce their intent to apply an operation by writing special shared variables, and the (single-threaded) recovery code analyzes the state of announced operations carefully after a failure to resolve their status. The outcome of this analysis indicates whether an operation took effect, including the operation's response if available, and is delivered back to the application also through shared variables. Since the application identifies each operation using a unique numerical ID, it is possible to ensure "exactly once" semantics of execution by retrying an operation, if needed, after a failure.

One of the open questions arising from [20] is how to formalize detectability as a correctness property of concurrent objects – a critical foundation for studying the complexity and computability of algorithmic problems related to such objects. We propose that a formal definition of detectability should address several desiderata: (D1) Detectability should be supported through the object's abstract interface, for example through specialized procedures. (D2) The definition should be independent of any particular model of computation or implementation style. (D3) In the context of shared memory, the definition should admit implementations on current generation hardware, which uses coherent but volatile caches (see *shared cache* model in [7]). The nesting-safe recoverable linearizability (NRL) framework of Attiya, Ben-Baruch, and Hendler [5], which we consider the current state of the art in this area of research, meets these goals only partially. In terms of properties (D1) and (D2), NRL mandates the use of concrete program variables for certain types of interaction with a detectable object, for example to receive helpful *auxiliary state* [6] from the system during recovery, and to record the response returned by a detectable operation. Thus, the object's interface is not entirely abstract, and NRL is somewhat specific to shared memory. Regarding property (D3), the modelling assumptions of the NRL framework do not account for the volatile cache, and impose the stringent requirement that a system recovering from failure can determine "the inner-most recoverable operation that was pending" for each process [5]. We are not aware of any practical implementation of NRL that realizes the latter behaviour.

The main contribution of this paper is a novel formal definition of detectability. Concretely, we introduce the *detectable sequential specification* (DSS), which augments the traditional method of specifying typed objects under sequential access with auxiliary procedures by which an application announces its intent to execute an operation that requires detectability, and resolves the outcome of this operation. The relationship between the DSS and prior work, specifically NRL [5] and its alternative definition (herein referred to as *NRL+*) proposed by Ben-David, Blelloch, Friedman and Wei in [7], can be summarized as follows:

1. The DSS specifies the behavior of detectable objects under sequential access only, and is used in tandem with an off-the-shelf correctness condition for concurrent objects (e.g., [2, 8, 24]). NRL instead uses conventional sequential specifications, and defines correct behavior under concurrent access in a unique way to simplify correct nesting of objects. Overall, our DSS-based approach comes closer to achieving properties (D1) and (D2), but delegates responsibility for nesting objects correctly to the application.
2. Although all three techniques use specialized recovery procedures to resolve the status of operations that may have been interrupted by failures, the semantics of these procedures are shaped by different goals. In DSS and NRL+, the recovery procedure allows a thread to *determine* whether or not an operation it intended to invoke prior to a failure took effect, and if so, what response it returned. In NRL, the purpose of the recovery procedure is to *ensure* that an invoked operation took effect, and determine its response.
3. Detectability in DSS is *declarative* in that the application calls a special *prepare procedure* to indicate which operations on a concurrent object must be detectable. Later on, an application may exercise its right to detectability by calling the recovery procedure, or not. In both NRL and NRL+, all operations are detectable. Furthermore, the recovery procedure in NRL is always invoked for an operation that was interrupted by a failure.
4. DSS-based objects require minimal system assumptions, and can be implemented using standard software tools on a current generation multiprocessor with persistent main memory and a volatile cache, thus achieving property (D3). NRL is based on a simplified *private cache* model [7] and relies fundamentally on auxiliary state [6]. NRL+ is similar to DSS in terms of system assumptions, but is formalized using unbounded sequence numbers to identify different operations, which complicates implementation.[1]

In summary, our DSS-based approach embeds detectability in a sequential specification, is implementable on today's multiprocessors without relying on a persistent call stack or multi-word failure-atomic writes, gives applications the unique ability to request detectability on demand, and leaves correct nesting of objects up to application code.

[1] In practice, sequence numbers are embedded in program variables, which reduces the number of bits available to store other state (e.g., a process ID and a data value in Algorithm 1 of [7]). This is especially problematic on current generation hardware, which supports only 64-bit failure-atomic writes [13, 41].

In addition to formalizing detectability, we also propose a novel detectable queue algorithm called the *DSS queue*. Our algorithm builds on the Michael and Scott queue [36] and its recoverable but non-detectable extension, the *durable queue* of Friedman, Herlihy, Marathe, and Petrank [20]. We compare the DSS queue experimentally (see Section 4) against several alternatives using a 20-core Intel multiprocessor with Optane persistent memory.

2 Detectable Sequential Specifications

Specifying and implementing detectability for shared objects is difficult in most models of computation precisely because modern computing systems store state using a combination of volatile and persistent media. As a result, a recovering process suffers from a mild case of amnesia that hinders forensic analysis – it cannot in general determine exactly what step it was about to perform at the point of failure, or what value was returned by its last operation on a given object. In the specific case of shared memory models with persistent memory, the use of explicit persistence instructions does not cure this ailment because the processor cannot both update a memory location and flush the new value to the "persistence domain" [41] in one atomic step. Following the approach of [5, 7], we treat the amnesia by prescribing the addition of specialized operations to the object's abstract interface.

Our goal in formalizing detectability in this section is to establish rigor, which is necessary for analyzing the complexity and computability of detectable shared objects, while defining the interface to such objects in an implementation-independent manner. We approach this task by extending the traditional approach of composing a sequential specification for an object's type T, which defines the object's correct behaviour under sequential access using data structure semantics (i.e., it prescribes a set of abstract states and state transitions), with a correctness property that describes correct behaviour under concurrent access [26]. The core idea is to augment a given type T with auxiliary operations by which processes declare their intent to execute a detectable operation, and then optionally resolve the status of this operation. This yields a detectable embodiment of T, which we denote as $D\langle T\rangle$. The sequential specification of type $D\langle T\rangle$, called the *detectable sequential specification (DSS)* of type T, is obtained automatically by a transformation of the original sequential specification of T, as explained shortly in Section 2.1. Finally, the correct behavior of a detectable object under concurrent access is formalized by composing $D\langle T\rangle$ with a correctness property suitable for a given model of computation (e.g., [2, 8, 24] in the shared memory context).

We present the formal definition of the DSS without a specific model of computation to emphasize that the DSS is largely model-agnostic. Sequential specifications in general are compatible with message passing, shared memory, and "m&m" [1] models. They accommodate both system-wide and individual process failure assumptions, and are orthogonal to assumptions regarding the volatility or persistence of storage media. The main modelling assumption required at this stage is the existence of a set Π of processes (or threads), where each process p_i has a distinct ID i. We assume implicitly that a process recovers under the *same* ID so that it can refer to its earlier actions that may have been interrupted by a failure.

2.1 Formal Definition

Consider an arbitrary object type T. Formally, T is a sequential specification denoted by a tuple $(S, s_0, OP, R, \delta, \rho)$ where S is a set of abstract states, $s_0 \in S$ is an initial state, OP is a set of operations (e.g., read(), write(1)), R is a set of possible operation responses, $\delta : S \times OP \times \Pi \to S$ is a state transition function indicating the effect of each operation by a process on the abstract state, and $\rho : S \times OP \times \Pi \to R$ is a response function indicating the operation's correct response.[2] A *detectable sequential specification (DSS)* for type $T = (S, s_0, OP, R, \delta, \rho)$, denoted $D\langle T\rangle$, is a sequential specification $(\bar{S}, \bar{s_0}, \bar{OP}, \bar{R}, \bar{\delta}, \bar{\rho})$ obtained from T by the following transformation:

- Each state $\bar{s} \in \bar{S}$ is a tuple $(s, \mathcal{A}, \mathcal{R})$ where $s \in S$, $\mathcal{A}$ is a mapping $\Pi \to OP \cup \{\bot\}$, and $\mathcal{R}$ is a mapping $\Pi \to R \cup \{\bot\}$, where $\bot \notin OP \cup R$.
- The initial state $\bar{s_o}$ is a tuple $(s_0, \mathcal{A}, \mathcal{R})$ where $\mathcal{A}$ and $\mathcal{R}$ map each element of Π to $\bot$.
- $\bar{OP}$ comprises all the operations of OP, as well as new *auxiliary operations*: *prep-op* and *exec-op* for each $op \in OP$, as well as *resolve*.
- $\bar{R} = R \cup \{(op, r) \mid op \in OP \cup \{\bot\} \wedge r \in R \cup \{\bot\}\}$
- The state transition function $\bar{\delta}$ and response function $\bar{\rho}$ for each operation $\bar{op} \in \bar{OP}$ are presented in Figure 1 using an axiomatic style modeled after [25, 26]. Each axiom indicates a pre-condition (first line), an operation / process ID / response (middle line), and a side-effect (third line). The latter indicates the new state using primed symbols; any component of the abstract state that is not explicitly referenced remains unchanged (e.g., Axiom 1 implies $s' = s$). Three of four axioms are parameterized by an operation $op \in OP$ of T, and yield a distinct operation $\bar{op} \in \bar{OP}$ of $D\langle T\rangle$ for each $op \in OP$.

$$\begin{array}{cc}
\begin{array}{c} \{true\} \\ \textit{prep-op} \;/\; p_i \;/\; \bot \\ \{\mathcal{A}'[p_i] = op \wedge \mathcal{R}'[p_i] = \bot\} \end{array} \quad (1) &
\begin{array}{c} \{\mathcal{A}[p_i] = op \wedge \mathcal{R}[p_i] = \bot\} \\ \textit{exec-op} \;/\; p_i \;/\; \rho(s, op, p_i) \\ \{s' = \delta(s, op, p_i) \wedge \mathcal{R}'[p_i] = \rho(s, op, p_i)\} \end{array} \quad (2) \\
\begin{array}{c} \{true\} \\ \textit{resolve} \;/\; p_i \;/\; (\mathcal{A}[p_i], \mathcal{R}[p_i]) \\ \{\} \end{array} \quad (3) &
\begin{array}{c} \{true\} \\ op \;/\; p_i \;/\; \rho(s, op, p_i) \\ \{s' = \delta(s, op, p_i)\} \end{array} \quad (4)
\end{array}$$

Figure 1 Detectable sequential specification (DSS) of type T, also denoted $D\langle T\rangle$.

In practical terms, the operations described axiomatically in Figure 1 behave as follows. For each $op \in OP$ of type T, *prep-op* (Axiom 1) and *exec-op* (Axiom 2) are used to declare the intention of a process p_i to apply op in a detectable way, and then apply it, respectively. Operation *prep-op* "remembers" op, and defines the context for a future call to *resolve* (Axiom 3), which determines the status of the most recently prepared operation. This *resolve* operation is somewhat similar to the recovery function introduced in NRL [5], but serves a different purpose: *resolve* is used to analyze the status of an operation that may have been left pending by a crash, whereas the recovery function always completes such an operation and returns its response. Finally, operation op (Axiom 4) simply applies the state transition prescribed by op in a non-detectable way with no other side-effects.

The DSS supports detectability in the following sense: after a call to *prep-op*, if *exec-op* took effect then *resolve* returns (op, r) where r is the response of op, otherwise it returns $(op, \bot)$. Since we assume that $\bot \notin R$, the response of *resolve* indicates to a process whether or not its execution of op via *exec-op* took effect. The *prep-op* and *resolve* operations are *total*, meaning that they can be called from any state, and *idempotent*, meaning that they can be called repeatedly (e.g., for example when their executions are interrupted by failures). If *prep-op* was never called for any op, then *resolve* returns $(\bot, \bot)$.

[2] The inclusion of the process ID in the arguments of δ and ρ is necessary since a detectable type encodes special recovery state for each process, and some of the operations query this state directly.

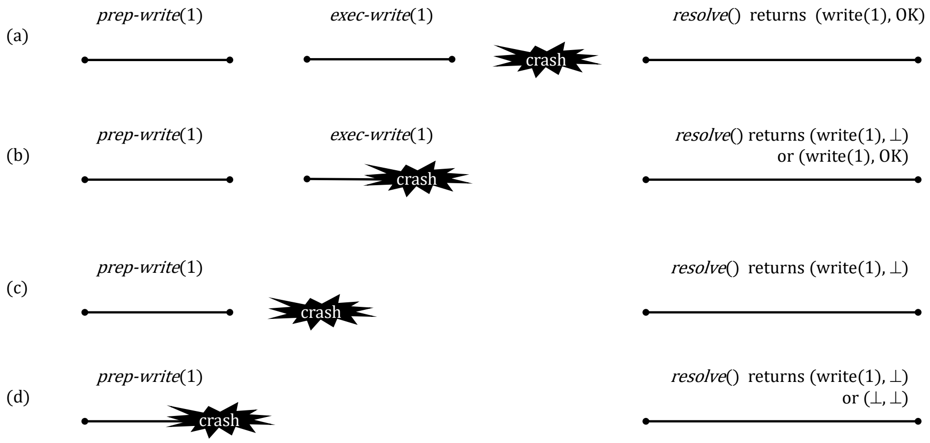


Figure 2 Informal examples of executions over an object that implements the DSS of a read/write register. The initial value of the register is 0, and time increases from left to right. Barbell symbols represent the time intervals of operation executions.

The state transitions of the DSS are illustrated in Figure 2, which presents four possible executions that can be generated by a detectable read/write register object. In example (a), a process p_i prepares a $write(1)$ operation, executes it, crashes, and resolves the operation as completed upon recovering. The mapping $\mathcal{A}$ records $write(1)$ as the prepared operation for the calling process p_i after $prep\text{-}write(1)$ takes effect, and $\mathcal{R}$ records OK as the response after $exec\text{-}write(1)$ takes effect. In (b), the crash occurs during $exec\text{-}write(1)$, and so the $write(1)$ state transition may or may not take effect. Thus, $resolve$ returns either $(write(1), \perp)$ or $(write(1), \mathrm{OK})$, and in both cases $\mathcal{A}[p_i]$ records $write(1)$ as the prepared operation. In (c), the crash occurs before the process invokes $exec\text{-}write$, and hence $resolve$ must return $(write(1), \perp)$. In (d), the crash occurs during $prep\text{-}write(1)$, and hence $resolve$ must return either $(\perp, \perp)$ to indicate that no operation was prepared, or $(write(1), \perp)$.

One special case deserving further attention occurs when a process p_i applies the same operation op repeatedly via $prep\text{-}op$ and $exec\text{-}op$, which makes the response of $resolve$ ambiguous. This problem can be remedied by augmenting the signature of op with an auxiliary argument that is saved in the state component $\mathcal{A}[p_i]$ but ignored in the computation of the state transition $\delta(s, op, p_i)$. For example, if the application is using a monotonic counter to record the number of detectable operations executed on the DSS-based object, then a single bit (i.e., the parity of the counter) is sufficient.

2.2 Discussion

The DSS-based approach marries the practical quality of Friedman, Herlihy, Marathe and Petrank's work [20] with the rigorous tone introduced by Attiya, Ben-Baruch, and Hendler's formalism [5]. More concretely, the DSS defines a purely object-oriented interface by which processes detect that status of past operations. Instead of relying explicitly on system support (as in NRL [5]) or on sequence numbers (as in NRL+ [7]) to identify an operation that may have been interrupted by a failure, the DSS internally records state regarding the last operation of each process via the auxiliary operation $prep\text{-}op$. The detection operation $resolve$ helps a recovering process identify the approximate position within its program where it crashed, for example distinguishing between cases (a) and (c) in Figure 2, and can replace the checkpointing mechanisms used in [5, 6] to some extent. A process may call the idempotent *resolve* operation arbitrarily many times to recover an earlier operation's response if its recovery efforts are hampered by additional crash failures. Such flexibility avoids having to save the response in a concrete program variable, which is how NRL deals with the problematic situation where a crash occurs immediately after an operation returns and before its response can be persisted.

For shared objects, the DSS must be combined with a suitable linearizability-like [26] correctness condition, and is compatible with several such conditions. In order from strongest to weakest, these include strict linearizability [2], persistent atomicity [24], and recoverable linearizability [8]. Note that the "program order inversion" anomaly in [8] only applies to operations on distinct objects, and cannot for example reorder an *exec-op* with a *resolve* on the same object. Our approach is inherently incompatible with the model underlying durable linearizability [30] because a crashed process in our framework must recover under the same ID to obtain meaningful output from operation *resolve*.[3] An analogous assumption is present in [5, 7].

A common misconception surrounding our work is that the DSS "does not support nesting." Indeed we do not prescribe a specific manner of recovering operations on nested objects (i.e., there is no "N" in DSS) because the DSS merely defines the abstract states and state transitions for a single object, but DSS-based objects can be nested, especially when they provide strict linearizability [2]. As an example, Section 3 of this paper describes an implementation of a DSS-based detectable queue from read/write register and Compare-And-Swap base objects in a shared memory model with persistent memory and volatile cache. Any base object of type T in this algorithm can be replaced with a strictly linearizable implementation of either T or $D\langle T\rangle$, since $D\langle T\rangle$ provides all the non-detectable operations of T. Thus, $D\langle\text{queue}\rangle$ can be constructed using implementations of $D\langle\text{read/write register}\rangle$ and $D\langle\text{CAS}\rangle$, and this demonstrates application-managed nesting of DSS-based objects. Finally, we point out that NRL [5] does not quite solve the problem of recovering nested objects either because much of the complexity associated with invoking recovery operations in the correct order is encapsulated in a crucial and difficult to implement system assumption.[4]

In terms of computability, the DSS-based approach is conveniently compatible with existing universal constructions of shared objects. For example, a wait-free recoverable implementation of $D\langle T\rangle$ for any conventional type T can be obtained in the shared memory model using Herlihy's universal construction [27], which was shown by Berryhill, Golab, and Tripunitara to yield recoverable linearizability in the presence of crash-recovery failures [8]. We believe that this construction can be extended easily from the "private cache" [7] model of persistent memory, where memory operations are assumed to persist immediately, to the more general model with volatile cache and explicit persistence instructions.

Little is known at this point regarding the complexity of DSS-based recoverable objects, but it is straightforward to show that such objects require linear space. Intuitively, this result holds across a variety of concrete models because the abstract state space of a DSS-based object encodes recovery information (via $\mathcal{A}$ and $\mathcal{R}$) for each process. NRL-like implementations also require linear space in some cases, as proved recently by Ben-Baruch, Hendler, and

[3] Durable linearizability [30] permits reuse of process IDs once the pending operations of crashed threads have "completed." We interpret this restriction to mean that both the pending operation and any detectability actions applied in connection with the operation have concluded.

[4] Section 2 of [5] states that "the system may eventually resurrect process p by invoking the recovery function of the inner-most recoverable operation that was pending when p failed."

Rusanovsky [6] for obstruction-free Compare-And-Swap. On the other hand, DSS-based objects behave very differently from NRL-like objects with respect to "auxiliary state," which some NRL-like objects receive from the application or from the system via specialized operation arguments (e.g., sequence numbers) or special shared variables. Whereas [6] proves that such external state must be provided to any NRL-like implementation of a "doubly-perturbing" type (which includes the FIFO queue), even with very weak non-blocking progress guarantees, one variation of the lock-free DSS queue algorithm presented in Section 3 requires no such state at all. Intuitively, this contrast follows from the fundamentally different semantics of recovery in DSS-based and NRL-like objects, where the former recover the most recently prepared (via a call to *prep-op*) operation and the latter recover the most recently invoked operation. Identifying the most recently invoked operation is inherently more difficult, and auxiliary state in NRL compensates for this difficulty.

3 A lock-free strictly linearizable detectable queue

As a proof of concept, we present in this section a DSS-based detectable queue implementation, called *DSS queue*, for the asynchronous shared memory model with persistent memory, volatile cache, and system-wide crash failures. The algorithm is based on Michael and Scott's venerable lock-free queue (called the *MS queue*) [36], as well as its recoverable variant for persistent memory (called the *durable queue*) published recently by Friedman, Herlihy, Marathe, and Petrank [20]. The original MS Queue uses a singly-linked list of nodes, referenced by *head* and *tail* pointers, to implement a FIFO queue, and is used heavily in practice (e.g., in the Java package java.util.concurrent). The durable queue adds the necessary flush instructions to cope with the volatile cache, and also augments the queue node structure by adding a $deqThreadID$ field, initially -1, to identify the thread who dequeues the value stored in the node. A queue node for which $deqThreadID \neq -1$ is called a *marked* node. The implementation assumes that a centralized recovery procedure is executed after each crash to complete pending operations and report their status to application threads using an array of shared variables called $returnedValues$.

We transform the n-thread durable queue into a DSS-based data structure by removing the $returnedValues$ array, adding an array $X[1..n]$ to represent the state components $\mathcal{A}$ and $\mathcal{R}$ of $D\langle queue\rangle$, and adding the auxiliary operations described in Section 2: *prep-op* and *exec-op* for each $op \in \{enqueue, dequeue\}$, as well as *resolve*. In the initial state, the *head* and *tail* pointers refer to the same sentinel node that is not marked, all entries of X are NULL, and every queue node has $next =$ NULL and $deqThreadID = -1$. The access procedures for the operations of $D\langle queue\rangle$ are presented using a syntax similar to C++, where & and $\rightarrow$ denote the usual reference and dereference operators. The keyword TID represents the identifier i of the calling thread t_i, where $1 \leq i \leq n$. Logical and bitwise AND, OR, and XOR are denoted exactly as in C++.

3.1 Enqueue and supporting operations

Enqueuing operations, presented in Figure 3, generally follow the code of durable queue [20], with the addition of operations to update the array X, which stores a pointer to a queue node. We borrow the most significant bits of this pointer to record tags that indicate whether or not the detectable *enqueue* operation was prepared and then took effect.[5]

[5] Modern x86-64 processors implement 48 address bits, which leaves 16 bits available for special tags.

Procedure *prep-enqueue*(*val*: value to be enqueued).

```
Node* $node$ := new Node($val$) // init: $next$ = NULL, $deqThreadID = -1$
FLUSH ($node$)
$X$[TID] := $node$ | ENQ_PREP_TAG
FLUSH (&$X$[TID])
```

Procedure *exec-enqueue*().

```
Node* $node$ := $X$[TID]
while true do
    Node* $last$ := $tail$
    Node* $next$ := $last \rightarrow next$
    if $last$ == $tail$ then
        if $next$ == NULL then // at tail
            if CompareAndSwap(&$last \rightarrow next$, NULL, $node$) then
                FLUSH (&$last \rightarrow next$)
                $X$[TID] := $X$[TID] | ENQ_COMPL_TAG
                FLUSH (&$X$[TID])
                CompareAndSwap (&$tail$, $last$, $node$)
                return
        else // help another enqueuing thread
            FLUSH (&$last \rightarrow next$)
            CAS(&$tail$, $last$, $next$)
```

Procedure *resolve*.

```
if $X$[TID] & ENQ_PREP_TAG then
    ($arg$, $ret$) := $resolve\text{-}enqueue$()
    return ($\langle enqueue, arg\rangle$ , $ret$)
else if $X$[TID] & DEQ_PREP_TAG then
    $ret$ := $resolve\text{-}dequeue$()
    return ($\langle dequeue,$ NO_ARG$\rangle$ , $ret$)
else // no operation was prepared
    return ($\perp, \perp$)
```

Procedure *resolve-enqueue*().

```
if $X$[TID] & ENQ_COMPL_TAG then
    // enqueue was prepared and took effect
    return (($X$[TID] ^ ENQ_PREP_TAG ^ ENQ_COMPL_TAG)$\rightarrow value$, OK)
else
    // enqueue was prepared and did not take effect
    return (($X$[TID] ^ ENQ_PREP_TAG)$\rightarrow value$, $\perp$)
```

Figure 3 The *prep-enqueue*, *exec-enqueue*, *resolve*, and *resolve-enqueue* operations of DSS queue.

The *prep-enqueue* operation creates a queue node that holds the value to be enqueued, and saves a pointer to the node along with a tag in X. The *exec-enqueue* operation follows closely the durable queue [20] algorithm by locating the tail of the linked list and swinging the next pointer of the tail node at line 11. The code at lines 13–14 updates the node pointer saved previously in X by setting the ENQ_COMPL_TAG and flushing the updated value, and is needed for detectability. The tail pointer is then updated at line 15. The code at lines 18–19 is a helping mechanism required for lock-freedom. The non-detectable *enqueue* operation is equivalent to calling *prep-enqueue* followed by *exec-enqueue*, except that any lines accessing X (3–4 and 13–14) are omitted.

The detection function *resolve* checks whether an *enqueue* was prepared at line 20, then calls a helper routine *resolve-enqueue* that considers two cases based on the presence of the ENQ_COMPL_TAG in X, which is added by *exec-enqueue* at lines 13–14, and also by the recovery procedure described later on. If the *enqueue* operation was prepared with value *val* and took effect, (*enqueue*(*val*), OK) is returned via lines 29 and 22. If the *enqueue* operation was prepared with value *val* and did not take effect, (*enqueue*(*val*), $\perp$) is returned via lines 31 and 22. Finally, if the *enqueue* operation was never prepared then either $(\perp, \perp)$ is returned at line 27 or a *dequeue* is resolved at lines 23–25.

3.2 Dequeue and supporting operations

The implementation of dequeuing operations is presented in Figure 4. Similarly to enqueuing operations, the code generally follows [20] with the addition of operations to update the array X, which stores a tagged pointer to a queue node. Two of the most significant bits of this pointer are repurposed to record a DEQ_PREP_TAG, which indicates whether or not the detectable *dequeue* was prepared, and an EMPTY_TAG, which indicates that a *dequeue* took effect on an empty queue.

The *prep-dequeue* operation initializes X using NULL tagged with DEQ_PREP_TAG. The *exec-dequeue* operation proceeds in two cases, the first of which follows closely the durable queue [20] algorithm. If the queue is found to be empty, meaning that the head and tail point to the same sentinel node where the *next* pointer is NULL at lines 38–40, then EMPTY_TAG is added to X at lines 41–42 and a special EMPTY value is returned at line 43. Otherwise the queue is not empty, and the correct return value is stored in the successor of the sentinel node at the head of the linked list. As in the durable queue, a thread tries to claim this value by using `CompareAndSwap` to write its ID into the *next* node at line 49, and if it succeeds, the updated *deqThreadID* field is then flushed at line 50. Next, the head pointer is advanced at line 51. On the other hand, if a competing thread causes the `CompareAndSwap` to fail, a helping mechanism is executed at lines 54–55 to persist the updated *deqThreadID* variable, and the code repeats. The main differences from the durable queue algorithm are two-fold. First, instead of returning the dequeued value using a dynamically allocated object, DSS queue returns it directly at line 52. Second, for detectability, the pointer to the head node is written to X with DEQ_PREP_TAG and flushed at lines 47–48. Overall, DSS queue performs one less flush in the helping mechanism due to the simplified method of returning the dequeued value, one more flush prior to each `CompareAndSwap` at line 48 due to detectability, and one less memory allocation.

The non-detectable *dequeue* operation is equivalent to calling *prep-dequeue* followed by *exec-dequeue*, with two differences. First, any lines accessing X (32–33, 41–42, and 47–48) are omitted. Second, to avoid confusion between a partially executed *exec-dequeue* and a *dequeue* during subsequent detection, the analog of line 49 in *dequeue* marks the *deqThreadID* field in a slightly different way: instead of using the caller's TID directly, it combines the TID with another special tag. Details are omitted due to lack of space.

Procedure *prep-dequeue*().

```
$X$[TID] := DEQ_PREP_TAG
FLUSH (&$X$[TID])
```

Procedure *exec-dequeue*().

```
while true do
    Node* $first$ := $head$
    Node* $last$ := $tail$
    Node* $next$ := $first \to next$
    if $first == head$ then
        if $first == last$ then // empty queue
            if $next ==$ NULL then // nothing new appended at tail
                $X$[TID] := $X$[TID] | EMPTY_TAG
                FLUSH (&$X$[TID])
                return EMPTY
            FLUSH (&$last \to next$)
            CompareAndSwap (&$tail, last, next$)
        else // non-empty queue
            $X$[TID] = $first$ | DEQ_PREP_TAG // save predecessor of node to
             be dequeued
            FLUSH (&$X$[TID])
            if CompareAndSwap (&$next \to deqThreadID$, $-1$, TID) then
                FLUSH (&$next \to deqThreadID$)
                CompareAndSwap (&$head, first, next$)
                return $next \to value$
            else if $head == first$ then // help another dequeuing thread
                FLUSH (&$next \to deqThreadID$)
                CompareAndSwap (&$head, first, next$)
```

Procedure *resolve-dequeue*().

```
if $X$[TID] == DEQ_PREP_TAG then
    // dequeue was prepared but did not take effect
    return $\perp$
else if $X$[TID] == (DEQ_PREP_TAG | EMPTY_TAG) then
    // empty queue
    return EMPTY
else if ($X$[TID] ^ DEQ_PREP_TAG)$\to next \to deqThreadID ==$ TID then
    // non-empty queue
    return ($X$[TID] ^ DEQ_PREP_TAG)$\to next \to value$
else
    // $X$ holds a node pointer, crashed before completing dequeue
    return $\perp$
```

Figure 4 The *prep-dequeue*, *exec-dequeue*, and *resolve-dequeue* operations of DSS queue.

The detection function *resolve* checks whether a *dequeue* was prepared at line 23, then calls a helper routine *resolve-dequeue* that determines the correct response by considering four cases based in part on the presence of a node pointer, DEQ_PREP_TAG and EMPTY_TAG in X. The latter is updated by *exec-dequeue* at lines 41–42, and 47–48, and also by the recovery procedure described later on. If X holds a NULL pointer with PREPARED_TAG only (line 56), this indicates that a *dequeue* was prepared but did not take effect, and so $\perp$ is returned at line 57. A NULL pointer with DEQ_PREP_TAG and EMPTY_TAG (line 58) indicates that a *dequeue* was prepared and took effect on an empty queue, and so EMPTY is returned at line 59. Otherwise, X stores a non-NULL node pointer with DEQ_PREP_TAG. If the *deqThreadID* field of the successor node matches the caller's TID (line 60), this indicates that a *dequeue* was prepared and took effect on a non-empty queue, and so *val* is returned at line 61 where *val* is the dequeued value obtained from the successor node. The final case (line 62) indicates a crash between line 47 and a successful `CompareAndSwap` at line 49, and so $\perp$ is returned at line 63 since no value was dequeued. In this case the successor node could have been marked by the same thread but in a non-detectable *dequeue*, by a different thread, or by no one.

3.3 Recovery

Discussion of recovery following a system crash is deferred to Appendix A due to lack of space. In summary, a single-threaded recovery procedure in the style of [20] adjusts $X[t_i]$ for each thread t_i after scanning the linked list of queue nodes to identify pending operations. The algorithm can be adapted straightforwardly to allow threads to recover independently, without relying on a centralized recovery phase, and this transformation eliminates the last trace of auxiliary state [6] from the DSS queue.

3.4 Analysis

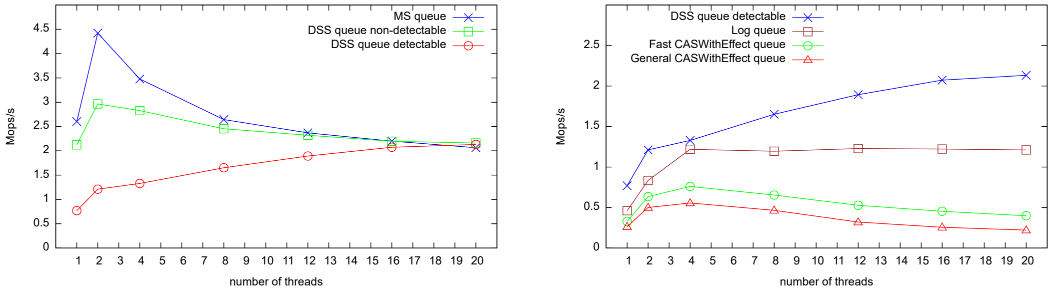
The correctness properties of the DSS queue algorithm are stated formally in Theorem 1. The analysis, including a detailed description of the model, is omitted due to lack of space.

▶ **Theorem 1.** *The DSS queue is lock-free and strictly linearizable with respect to $D\langle queue\rangle$.*

4 Evaluation

In this section, we present an empirical evaluation of the DSS queue algorithm from Section 3. The experiments are conduced using a 20-core 2.1GHz Intel Xeon processor equipped with genuine Intel Optane Data Center Persistent Memory Modules (DCPMM) provisioned in App Direct mode. Turbo boost is disabled to reduce variation in running times. The software environment includes Ubuntu Linux 20.04, g++ 9.3.0, and version 1.8 of the Intel Persistent Memory Development Kit (PMDK) library [42]. The code is compiled in debug mode with optimization level O0. The Optane memory is accessed using standard C++ atomic operations configured with sequentially consistent ordering. We flush data from the volatile cache to the Optane device using the PMDK *pmem_persist* function, which internally uses Intel's cache line write back (CLWB) instruction and also includes a store fence. For memory management, each thread pre-allocates a fixed size pool of queue nodes at initialization, and dequeued nodes are returned to the free pool using epoch-based reclamation (EBR) [17]. The EBR code is borrowed from the open-source implementation (`https://github.com/microsoft/pmwcas`) of Wang et al.'s Persistent Multi-word Compare-And-Swap (PMwCAS) [45]. The recovery procedure described in Appendix A is extended straightforwardly to prevent memory leaks, such as due to a crash in *prep-enqueue*.

Our experiments, presented in Figure 5, evaluate the scalability of different queue implementations in failure-free runs with up to 20 threads. In each experiment, the queue is initialized with 16 queue nodes, and each thread executes alternating pairs of *enqueue* and *dequeue* operations for 30 seconds. Each point plotted in the graphs is the mean throughput value (millions of operations per second) computed over a sample of ten runs, and in all cases the sample standard deviation is less than 2% of the sample mean.



(a) Different levels of detectability and persistence

(b) Different detectable queue implementations

Figure 5 Scalability experiments.

Figure 5a investigates the cost of detectability, which threads can request on demand in our flexible DSS-based approach. The three points of comparison in Figure 5a are the following queue implementations:

- *DSS queue detectable*: the DSS queue algorithm as described in Section 3, where *enqueue* and *dequeue* are applied in a detectable manner via calls to *prep-enqueue*/*exec-enqueue* and *prep-dequeue*/*exec-dequeue*. Procedures *resolve* and *resolve* are not invoked since we consider only failure-free runs.
- *DSS queue non-detectable*: the DSS queue algorithm where *enqueue* and *dequeue* are applied in a non-detectable manner.
- *MS queue*: an implementation of the classic MS queue [36] obtained from the non-detectable DSS queue by removing flushes in *enqueue* and *dequeue*.

The scalability plot shows that non-detectable implementation is measurably faster than detectable, offering nearly 3× higher throughput at low levels of parallelism. This is primarily due to the cost of the memory operations at lines 3-4, 13–14, 32–33, and 47–48 in the detectable execution path. The additional latency due to these operations has a smaller effect on throughput at higher levels of parallelism, and performance is ultimately limited by the synchronization bottleneck at the head and tail of the queue. The MS queue has the best performance overall and beats DSS non-detectable by up to 1.5× at 2 threads, and similarly suffers from contention at higher levels of parallelism. Indeed, all three scalability curves nearly converge at 20 threads.

Figure 5b compares the DSS queue with several other detectable queue algorithms:

- *DSS queue*: our DSS queue algorithm, identical to "DSS queue detectable" in Figure 5a.
- *General CASWithEffect queue*: a simple queue algorithm where the linked list and detectability state (analogous to X in DSS queue) are manipulated using the Persistent Multi-word Compare-And-Swap (PMwCAS) algorithm of Wang et al. [45].

- *Fast CASWithEffect queue*: similar to General CASWithEffect queue, except that PMwCAS is optimized for multi-word operations that access a combination of shared variables (queue head, tail, and next pointers) and private variables (detectability state).
- *Log queue*: our own implementation of Friedman et al.'s detectable *log queue* algorithm [20], which uses per-thread logs. Operation arguments and return values are stored directly in the logs, and are accessed by other threads via helping mechanisms.

The results show that DSS queue provides superior scalability to the two algorithms based on powerful PMwCAS, which simplifies the implementation greatly but becomes a performance bottleneck as contention rises. Furthermore, Fast CASWithEffect queue outperforms General CASWithEffect queue by up to 1.5×, showing the benefit of optimizing multi-word operations that access private variables atomically with shared variables. Our implementation of the log queue also outperforms both Fast and General CASWithEffect queue, but trails behind DSS queue by up to 1.7×. One reason why DSS queue is faster is that it stores detectability state (array X) using variables that are statically allocated and effectively private as they are shared only with the single-threaded recovery procedure. In comparison, the log queue dynamically allocates log objects in addition to queue nodes, and these objects are shared during concurrent execution of *dequeue*. We plan in future work to compare DSS queue against a more heavily optimized implementation of the log queue, which may narrow the observed performance gap.

5 Related Work

Research on software techniques for persistent memory, also known as non-volatile RAM (NVRAM), erupted roughly a decade ago as solid-state secondary storage devices began to flood the market and hardware vendors began to plan for production of high-capacity byte-addressable persistent main memory modules. Several software frameworks were proposed around that time to provide low-level access to persistent state through file systems, persistent heaps, and lightweight transactions based on redo and undo logging [10, 12, 44]. In parallel with these efforts, early designs of durable data structures for persistent memory began to emerge (e.g., [43]). Subsequent practical work introduced a variety of performance optimizations based on judicious use of memory fences, persistence instructions, and hybrid memory hierarchies that combine both volatile and non-volatile main memories [9, 14, 15, 29, 31, 35, 37, 40].

More recent publications describe the design of specific in-memory data structures and synchronization objects for persistent memory, and give up the convenience of transaction-based implementations for the sake of better performance through specialized concurrency control and persistence mechanisms. Much of this work focuses on practical scalable index structures for databases, such as hash maps and search trees, some of which exploit hybrid memory hierarchies [3, 16, 34, 38, 39, 45, 46]. Due to the inherent difficulty of designing and verifying the correctness such structures, researchers have also sought efficient (i.e., non-transactional) transformations of conventional in-memory data structures to durable structures for persistent memory [18, 33], which are applicable only when the original structure follows certain common design patterns, and hence not universal. In terms of synchronization objects, several non-blocking implementations of queues, read/write registers, Compare-And-Swap, and consensus objects are known [5, 6, 7, 8, 20, 21]. Wait-free [8] and lock-free [11] universal constructions have also been proposed for such objects.

Several attempts have been made to formalize the correctness properties of concurrent objects for persistent memory through variations on Herlihy and Wing's widely-adopted linearizability property [26] and Lamport's atomic register [32]. In one line of work, which assumes that processes or threads recover after a crash and continue execution under the same identifier, the problem was already solved earlier in the context of message passing systems by the strict linearizability property of Aguilera and Frølund [2] and persistent atomicity property of Guerraoui and Levy [24]. Berryhill, Tripunitara, and Golab [8] later relaxed these conditions by proposing recoverable linearizability, and formalized recoverable objects in the shared memory model. Izraelevitz, Mendes and Scott proposed an alternative model where thread identifiers are not reused (at least not immediately) after a crash, which makes persistent atomicity, recoverable linearizability, and ordinary linearizability indistinguishable, and defined durable linearizability as a synonym for this merged correctness condition [30]. They also introduce an innovative property called buffered durable linearizability that allows applications to trade durability guarantees for better performance due to reduced use of costly persistence instructions.

The question of whether process or thread identifiers are reused across crashes has been debated in the community. On one hand, it is true that from the point of view of the operating system, a thread resurrected after a crash is distinct from any thread that ran prior to the failure. On the other hand, applications tend to number their threads internally using simple schemes (e.g., 1 to n), and this establishes a secondary identity that survives crash failures. Such internal identifiers are used extensively in the implementations of synchronization objects, for example to index arrays. Curiously, we observe this pattern even in some durably linearizable implementations [6, 20].

Detectability was introduced informally by Friedman, Herlihy, Marathe and Petrank [19, 20], and formalized by Attiya, Ben-Baruch, and Hendler as nesting-safe recoverable linearizability (NRL) [5]. NRL is stronger than our DSS (Section 2) in the sense that it specifies precisely how to recover nested implementations, whereas DSS embodies detectability only. DSS-based objects can be nested in principle, and strictly linearizable implementations of such objects can be used in place of atomic base objects if preservation of probabilistic properties [22] is not a concern. Both NRL and the DSS augment the interface of concurrent objects with special recovery procedures, which allows a recovering thread to resolve its own pending operation after a failure, and in that sense implies the reuse of thread identifiers in the spirit of [2, 8, 24]. In contrast, pending operations are resolved by a separate recovery process in [20], which makes their approach to detectability compatible with the durable linearizability.

In terms of complexity, prior work has established bounds on the number of persistent fence instructions required by deterministic lock-free durably linearizable implementations [11], and the space required by NRL-like implementations [6]. It is also known that NRL-like implementations require some form of auxiliary state from the system [6]. Our DSS queue algorithm is exempt from the latter impossibility result, but one of its variations relies on auxiliary state in the sense that the system must initiate the centralized recovery phase (as in [20]), which updates some of the queue's base objects. Another variation of the DSS queue avoids the centralized recovery phase and has no dependence at all on auxiliary state.

6 Conclusion

In this paper, we introduced the detectable sequential specification (DSS) as a formal definition of detectability for recoverable objects. As a proof of concept, we presented a DSS-based recoverable queue algorithm, and evaluated its performance on a multiprocessor with Intel Optane persistent memory. We hope that the DSS opens a new avenue for both rigorous analysis and practical implementation of recoverable concurrent objects.

References

1 Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–60, 2018.

2 Marcos K. Aguilera and S. Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

3 Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, 2018.

4 Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in order to recover: Recoverable lock-free data structures. *CoRR*, abs/1905.13600, 2019. `arXiv:1905.13600`.

5 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, 2018.

6 Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 11–20, 2020.

7 Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

8 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

9 Trevor Brown and Hillel Avni. Phytm: Persistent hybrid transactional memory. *Proc. VLDB Endow.*, 10(4):409–420, 2016.

10 Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, 2011.

11 Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. The inherent cost of remembering consistently. In *Proc. of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 259–269, 2018.

12 Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, 2009.

13 Intel Corporation. Persistent memory faq, 2020. [last accessed 2/17/2021]. URL: `https://software.intel.com/content/www/us/en/develop/articles/persistent-memory-faq.html`.

14 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proc. of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, 2018.

15 Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proc. of the 15th EuroSys Conference*, pages 5:1–5:15, 2020.

16 Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC)*, pages 373–386, 2018.

17 Keir Fraser. *Practical Lock-Freedom.* PhD thesis, University of Cambridge, 2004. Computer Laboratory.

18 Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: in NVRAM data structures, the destination is more important than the journey. In *Proc. of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 377–392, 2020.

19 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. Brief announcement: A persistent lock-free queue for non-volatile memory. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, volume 91, pages 50:1–50:4, 2017.

20 Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, 2018.

21 Wojciech Golab. The recoverable consensus hierarchy. In *Proc. of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 281–291, 2020.

22 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *In Proc. of the 43rd ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 2011.

23 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.

24 Rachid Guerraoui and Ron R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407, 2004.

25 John V. Guttag, James J. Horning, and Jeannette M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, 1985.

26 M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

27 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

28 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

29 Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proc. of the 21s International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, 2016.

30 Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proc. of the 30th International Symposium on Distributed Computing (DISC)*, pages 313–327, 2016.

31 Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *ACM SIGPLAN Notices*, 51(4):385–398, 2016.

32 L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

33 Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *Proc. of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 462–477, 2019.

34 Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proc. VLDB Endow.*, 13(4):574–587, 2019.

35 Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, 52(4):329–343, 2017.

36 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

37 Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proc. of the 18th International Conference on Extending Database Technology (EDBT)*, pages 689–694, 2015.

38 Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, pages 37:1–37:16, 2017.

39 Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory. In *Proc. of the 2016 International Conference on Management of Data (SIGMOD)*, pages 371–386. ACM, 2016.

40 Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *IEEE Micro*, 35(3):125–131, 2015.

41 Andy Rudoff. Persistent memory programming. *login Usenix Mag.*, 42(2), 2017.

42 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit, 2020. [last accessed 2/11/2021]. URL: https://pmem.io/pmdk/.

43 Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

44 Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, 2011.

45 Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy lock-free indexing in non-volatile memory. In *Proc. of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, 2018.

46 Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Trans. Computers*, 65(7):2169–2183, 2016.

A Recovery of DSS queue

Following [20], we assume that a recovery phase is executed after each crash, before the application threads are revived. This is an implementation detail, and is not required in general for correctness of a DSS-based implementation. In practical terms, the recovery phase is the execution of a recovery procedure by the main thread of the application. As in [20], the recovery code scans the linked list of queue nodes from the head pointer and checks the value of the *deqThreadID* field. For any marked node, the original algorithm ensures completion of the *dequeue* operation by recording the dequeued value in a special array of shared variables, but this is not required in our algorithm. This is because we do not use shared variables to return dequeued values, and the state required for detectability is already established at lines 47–48 of *exec-dequeue* prior to calling `CompareAndSwap` at line 49. Next, the recovery code advances the *head* pointer, if needed, to point to the last marked node reachable from the current head pointer, and sets the tail pointer to the last node in the linked list. Finally, we extend the recovery algorithm to detect any pending *enqueue* operations that took effect in the sense of persisting an updated *next* pointer at line 12, but did not record this state change for detectability (in our case at lines 13–14). During the traversal of the linked list, if a node is encountered that is referenced by some element of X and where the pointer is tagged with ENQ_PREP_TAG, the code tags the pointer with ENQ_COMPL_TAG as well; this takes care of nodes that were enqueued and had not yet been dequeueud at the point of failure. For nodes that were first enqeued and then dequeued, the code checks the pointers in X that are tagged with ENQ_PREP_TAG, and sets ENQ_COMPL_TAG for any marked node. The complete algorithm is presented in Figure 6.

Procedure *recovery*().

```
$AllNodes$ := set of queue nodes reachable from $head$
$tail$ := last queue node reachable from $head$
FLUSH ($\&tail$)
$oldHead$ := $head$
$head$ := last marked node reachable from $oldHead$
FLUSH ($\&head$)
for $i \in 1..n$ do
    if $X[i]$ is a pointer to a node $d \in AllNodes$ and is tagged with ENQ_PREP_TAG
     but not ENQ_COMPL_TAG then
        // enqueued and still in the linked list
        $X[i] := X[i]$ | ENQ_COMPL_TAG
        FLUSH ($\&X[i]$)
    else if $X[i]$ is a pointer to a node $d \notin AllNodes$ and is tagged with
     ENQ_PREP_TAG but not ENQ_COMPL_TAG, and $d \rightarrow deqThreadID \neq -1$
     then
        // enqueued and no longer in the linked list, already marked
        $X[i] := X[i]$ | ENQ_COMPL_TAG
        FLUSH ($\&X[i]$)
```

Figure 6 Recovery procedure of DSS queue.

Constant RMR Group Mutual Exclusion for Arbitrarily Many Processes and Sessions

Liat Maor ✉
The Interdisciplinary Center, Herzliya, Israel

Gadi Taubenfeld ✉
The Interdisciplinary Center, Herzliya, Israel

Abstract

Group mutual exclusion (GME), introduced by Joung in 1998, is a natural synchronization problem that generalizes the classical mutual exclusion and readers and writers problems. In GME a process requests a session before entering its critical section; processes are allowed to be in their critical sections simultaneously provided they have requested the same session.

We present a GME algorithm that (1) is the first to achieve a constant Remote Memory Reference (RMR) complexity for both cache coherent and distributed shared memory machines; and (2) is the first that can be accessed by arbitrarily many dynamically allocated processes and with arbitrarily many session names. Neither of the existing GME algorithms satisfies either of these two important properties. In addition, our algorithm has constant space complexity per process and satisfies the two strong fairness properties, first-come-first-served and first-in-first-enabled. Our algorithm uses an atomic instruction set supported by most modern processor architectures, namely: read, write, fetch-and-store and compare-and-swap.

2012 ACM Subject Classification Theory of computation; Theory of computation → Distributed computing models; Theory of computation → Shared memory algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Group mutual exclusion, RMR complexity, unbounded number of processes, fetch&store (FAS), compare&swap (CAS)

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.30

Acknowledgements We thank the anonymous referees for their constructive suggestions.

1 Introduction

1.1 Motivation and results

In the *group mutual exclusion* (GME) problem n processes repeatedly attend m sessions. Processes that have requested to attend the same session may do it concurrently. However, processes that have requested to attend different sessions may not attend their sessions simultaneously. The GME problem is a natural generalization of the classical mutual exclusion (ME) and readers/writers problems [9, 12]. To see this, observe that given a GME algorithm, ME can be solved by having each process uses its unique identifier as a session number. Readers/writers can be solved by having each writer requests a different session, and having all readers request the same special session. This allows readers to attend the session concurrently while ensuring that each writer attends in isolation. The GME problem has been studied extensively since it was introduced by Yuh-Jzer Joung in 1998 [21, 22].

A simple example has to do with the design of a concurrent queue or stack [6]. Using a GME algorithm, we can guarantee that no two users will ever simultaneously be in the *enqueue.session* or *dequeue.session*, so the enqueue and dequeue operations will never be interleaved. However, it will allow any number of users to be in either the enqueue or

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 30; pp. 30:1–30:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

dequeue session simultaneously. Doing so simplifies the design of a concurrent queue as our only concern now is to implement concurrent enqueue operations and concurrent dequeue operations.

In this paper, we present a GME algorithm that is the first to satisfy several desired properties (the first two properties are satisfied only by our algorithm).

1. *Suitability for dynamic systems*: All the existing GME algorithms are designed with the assumption that either the number of processes or the number of sessions is a priori known. Our algorithm is the first that does not make such an assumption:
   - it can be accessed by an arbitrary number of processes; that is, processes may appear or disappear intermittently, and
   - the number and names of the sessions are not limited in any way.
2. $O(1)$ *RMR complexity*: An operation that a process performs on a memory location is considered a remote memory reference (RMR) if the process cannot perform the operation locally on its cache or memory and must transact over the multiprocessor's interconnection network in order to complete the operation. RMRs are undesirable because they take long to execute and increase the interconnection traffic. Our algorithm
   - achieves the ideal RMR complexity of $O(1)$ for Cache Coherent (CC) machines; and
   - is the first to achieve the ideal RMR complexity of $O(1)$ for Distributed Shared Memory (DSM) machines. (In Subsection 1.3, we explain why this result does not contradict the lower bound from [11].)

   This means that a process incurs only a constant number of RMRs to satisfy a request (i.e, to enter and exit the critical section once), regardless of how many other processes execute the algorithm concurrently.
3. $O(1)$ *space per process*: A small constant number of memory locations are allocated for each process. On DSM machines, these memory locations reside in the process local memory; on CC machines, these locations reside in the shared memory.
4. *Strong fairness*: Requests are satisfied in the order of their arrival. That is, our algorithm satisfies the *first-come-first-served* and *first-in-first-enabled* properties, defined later.
5. *Hardware support*: Atomic instruction set that is supported by most modern processor architectures is used, namely: read, write, fetch-and-store and compare-and-swap.

We point out that when using a GME as a ME algorithm, the number of processes is the same as the number of sessions (each process uses its identifier as its session number). Thus, in GME algorithms, in which the number of sessions is a priori known also the number of processes must be known, at least when these GME algorithms are used as ME algorithms or readers and writers locks.

Our GME algorithm is inspired by J. M. Mellor-Crummey and M. L. Scott MCS queue-based ME algorithm [28]. The idea of our GME algorithm is to employ a queue, where processes insert their requests for attending a session. The condition when a process p may attend its session depends on whether p's session is the same as that of all its predecessors. Otherwise, p waits until p is notified (by one of its predecessors) that all its predecessors which have requested different sessions completed attending their sessions.

A drawback of the MCS ME algorithm is that releasing a lock requires spinning – a process p releasing the lock may need to wait for a process that is trying to acquire the lock (and hence is behind p in the queue) to take a step before p can proceed. The ME algorithms in [13] overcome this drawback while preserving the simplicity, elegance, and properties of the MCS algorithm. We use a key idea inspired by [13] in our GME algorithm to ensure that a process releasing the GME lock will never have to wait for a process that has not attended its session yet.

Another key idea of our algorithm is to count down completed requests for attending a session by moving a pointer by one node (in the queue) for each such request and to ensure the integrity of this scheme by gating the processes that have completed attending a session (and are now trying to move the pointer) through a mutual exclusion lock.

1.2 The GME problem

More formally, the GME problem is defined as follows: it is assumed that each process executes a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section (CS)*, and *exit*.

A process starts by executing its remainder section. At some point, it might need to attend some session, say s. To attend session s, a process has to go through an entry code that guarantees that while it is attending this session, no other process is allowed to attend another session. In addition, once a process completes attending a session, the process executes its exit section in which it notifies other processes that it is no longer attending the session. After executing its exit section, the process returns to its remainder.

The group mutual exclusion problem is to write the code for the *entry section* and the *exit section* so that the following requirements are satisfied.

- *Mutual exclusion*: Two processes can be in their CS at the same time, only if they request the same session.
- *Starvation-freedom*: If a process is trying to enter its CS, then this process must eventually enter its CS.
- *Group concurrent entering* (*GCE*): If a process p requests a session s while no process is requesting a conflicting session, then (1) some process with session s can complete its entry section within a bounded number of its own steps, and (2) p eventually completes its entry section, even if other processes do not leave their CS.
- *Group bounded exit* (*GBE*): If a process p is in its exit section, then (1) some process can complete its exit section within a bounded number of its own steps, and (2) p eventually completes its exit section.

GCE precludes using a given mutual exclusion algorithm as a solution for the GME problem since GCE enables processes to attend the same session concurrently.

Our algorithm also satisfies the following strong fairness requirements. To formalize this, we assume that the entry code starts with a bounded section of code (i.e., one that contains no unbounded loops), called the *doorway*; the rest of the entry code is called the waiting room. The fairness requirements, satisfied by our algorithm, can now be stated as follows:

- *First-come-first-served* (*FCFS*): If a process p completes its doorway before a process q enters its doorway and the two processes request different sessions, then q does not enter its CS before p enters its CS [16, 26].
- *First-in-first-enabled* (*FIFE*): If a process p completes its doorway before a process q enters its doorway, the two processes request the same session, and q enters its CS before p, then p enters its CS in a bounded number of its own steps [20].

We notice that FCFS and FIFE do not imply starvation-freedom or group concurrent entering.

1.3 Further explanations

To illustrate the various GME requirements, imagine the critical section as a lecture hall that different professors can share for their lectures. Furthermore, assume that the lecture hall has one entrance door and one exit door. When solving the GME problem, the property of mutual exclusion guarantees that two different lectures cannot be arranged in the lecture hall simultaneously, while starvation-freedom guarantees that the lecture hall will eventually be reserved for every scheduled lecture.

Assuming that only one lecture is scheduled, group concurrent entering ensures that all the students who want to attend this lecture can enter the lecture hall through the entrance door, possibly one after the other, and attend the lecture. Furthermore, at any given time, when there are students who want to attend the lecture, at least one of them can always enter the lecture hall without any delay. Similarly, group bounded exit ensures that all the students who want to leave a lecture can do so through the exit door, possibly one after the other. Furthermore, at any given time, at least one of them can exit the lecture hall without delay.

Group concurrent entering and group bounded exit are first introduced and formally defined in this paper. They are slightly weakened versions of two known requirements (formally defined below) called concurrent entering and bounded exit. Using the lecture hall metaphor, assuming that only one lecture is scheduled, concurrent entering ensures that all the students who want to attend this lecture can enter the lecture hall *together*. Similarly, bounded exit ensures that all the students who want to leave a lecture can do so *together*. So, why have we not used these two stronger requirements?

Danek and Hadzilacos lower bound. Let n denotes the total number of processes. In [11], it is proven that $\Omega(n)$ RMRs are required for any GME algorithm that satisfies mutual exclusion, starvation-freedom, concurrent entering, and bounded exit, in the DSM model, using basic primitives of any strength. This result holds even when the number of sessions is only two. (Concurrent entering and bounded exit are as defined below [16].) Since we are aiming at finding a solution that has $O(1)$ RMR complexity, we had to weaken either concurrent entering, bounded exit, or both. (GME would not be interesting if the mutual exclusion or starvation-freedom properties are weakened.)

Group concurrent entering. To avoid an inefficient solution to the GME problem using a traditional ME algorithm and forcing processes to be in their CS one-at-a-time, even if all processes are requesting the same session, Joung required that a GME algorithm satisfies the following property (which he called concurrent entering):

- If some processes request a session and no process requests a different session, then the processes can concurrently enter the CS [21].

The phrase “can concurrently enter,” although suggestive, is not precise. In [24, 25], Keane and Moir were the first to give a precise definition that captures their interpretation of Joung’s requirement (which they also called concurrent entering):

- *Concurrent occupancy*: If a process p requests a session and no process requests a different session, then p eventually enters its CS, even if other processes do not leave their CS. (The name “concurrent occupancy” is from [16].)

In [16], Hadzilacos gave the following interpretation, which is stronger than that of Keane and Moir.

- *Concurrent entering*: If some process, say p, is trying to attend a session s while no process is requesting a conflicting session, then p completes its entry section in a bounded number of its own steps.

To circumvent the Danek and Hadzilacos $\Omega(n)$ lower bound, we looked for a slightly weaker version of concurrent entering that would still capture the property that Joung intended to specify. We believe that group concurrent entering, which is strictly stronger than concurrent occupancy, is such a property. We point out that our algorithm actually satisfies the following stronger version of group concurrent entering,

- *Strong group concurrent entering*: If a process p requests a session s, and p completes its doorway before any conflicting process starts its doorway, then (1) some process with session s can complete its entry section within a bounded number of its own steps, and (2) p eventually completes its entry section, even if other processes do not leave their CS.

Strong group concurrent entering (SGCE) is a slightly weakened version of a known property called strong concurrent entering [20].

Group bounded exit. Our group bounded exit property is replaced by the following two (weaker and stronger) properties in previously published papers.

- *Terminating exit*: If a process p enters its exit section, then p eventually completes it [24].
- *Bounded exit*: If a process p enters its exit section, then p eventually completes it within a bounded number of its own steps [16].

Again, to circumvent the Danek and Hadzilacos $\Omega(n)$ lower bound, we have defined group bounded exit, which is slightly weaker than bounded exit and is strictly stronger than terminating exit.

Open question. We have modified both concurrent entering and bounded exit. Is this necessary? With minor modifications to the Danek and Hadzilacos lower bound proof, it is possible to prove that their lower bound still holds when replacing only bounded exit with group bounded exit. Thus, to circumvent the lower bound, the weakening of concurrent entering is necessary. However, the question of whether it is possible to circumvent the lower bound by replacing only concurrent entering with group concurrent entering, and leaving bounded exit as is, is open.

1.4 Related work

Table 1 summarizes some of the (more relevant) GME algorithms mentioned below and their properties. The group mutual exclusion problem was first stated and solved by Yuh-Jzer Joung in [21, 22], using atomic read/write registers. The problem is a generalization of the mutual exclusion problem [12] and the readers and writers problem [9] and can be seen as a special case of the drinking philosophers problem [8].

Group mutual exclusion is similar to the room synchronization problem [6]. The room synchronization problem involves supporting a set of m mutually exclusive "rooms" where any number of users can execute code simultaneously in any one of the rooms, but no two users can simultaneously execute code in separate rooms. In [6], room synchronization is defined using a set of properties that is different than that in [21], a solution is presented, and it is shown how it can be used to efficiently implement concurrent queues and stacks.

In [24, 25], a technique of converting any solution for the mutual exclusion problem to solve the group mutual exclusion problem was introduced. The algorithms from [24, 25] do not satisfy group concurrent entering and group bounded exit and have $O(n)$ RMR complexity, where n is the number of processes. (By a mistake, in some of the tables in [24, 25], smaller RMR complexity measures are mentioned.) In [16], a simple formulation of concurrent entering is proposed which is stronger than the one from [24], and an algorithm is presented that satisfies this property.

In [20], the first FCFS GME algorithm is presented that uses only $O(n)$ bounded shared registers, while satisfying concurrent entering and bounded exit. Also, it is demonstrated that the FCFS property does not fully capture the intuitive notion of fairness, and additional fairness property, called first-in-first-enabled (FIFE) was presented. Finally, the authors presented a reduction that transforms any *abortable* FCFS mutual exclusion algorithm, into a GME algorithm, and used it to obtained GME algorithm satisfying both FCFS and FIFE.

A GME algorithm is presented in [11] with $O(n)$ RMR complexity in the DSM model, and it is proved that this is asymptotically optimal. Another algorithm in [11] requires only $O(\log n)$ RMR complexity in the CC model, but can be used just for two sessions.

Our algorithm satisfies FCFS fairness. That is, if the requests in the queue are for sessions 1, 2, 1, 2, 1, 2 and so on, those requests would be granted in that order. Yet, for practical considerations, one may want to batch all requests for session 1 (and, separately, for session 2) and run them concurrently. Our algorithm does not support "batching" of pending requests for the same session, as FCFS fairness and "batching" of pending requests for the same session are contradicting (incompatible) requirements. This idea was explored in [5], where a GME algorithm is presented that satisfies two "batching" requirements call pulling and relaxed-FCFS, and requiring only $O(\log n)$ RMR complexity in the CC model. Reader-Writer Locks were studied in [7], which trade fairness between readers and writers for higher concurrency among readers and better back-to-back batching of writers.

An algorithm is presented in [14] in which a process can enter its critical section within a constant number of its own steps in the absence of any other requests (which is typically referred to as contention-free step complexity). In the presence of contention, the RMR complexity of the algorithm is $O(min(k, n))$, where k denotes the interval contention. The algorithm requires $O(n^2)$ space and does not satisfy fairness property like FCFS or FIFE.

In [2], a GME algorithm with a constant RMR complexity in the CC model is presented. This algorithm does not satisfy group concurrent entering (or even concurrent occupancy) and FCFS. However, it satisfies two other interesting properties (defined by the authors) called simultaneous acceptance and forum-FCFS.

In [17], the first GME algorithm with both linear RMR complexity (in the CC model) and linear space was presented, which satisfies concurrent entering and bounded exit, and uses only read/write registers. A combined problem of ℓ-exclusion and group mutual exclusion, called the group ℓ-exclusion problem, is considered in [29, 32].

Besides the algorithms mentioned above, for the shared-memory model, there are algorithms that solve the GME problem under the message-passing model. Several types of the network's structure were considered, for example, tree networks [4], ring networks [33], and fully connected networks [3]. In [3, 23, 31], quorum-based message-passing algorithms are suggested in which a process that is interested in entering its CS has to ask permission from a pre-defined quorum.

2 Preliminaries

2.1 Computational model

Our model of computation consists of an asynchronous collection of n deterministic processes that communicate via shared registers (i.e., shared memory locations). Asynchrony means that there is no assumption on the relative speeds of the processes. Access to a register is done by applying operations to the register. Each operation is defined as a function that gets as arguments one or more values and registers names (shared and local), updates the value of the registers, and may return a value. Only one of the arguments may be a name of a *shared* register. The execution of the function is assumed to be atomic. Call by reference is used when passing registers as arguments. The operations used by our algorithm are:

- *Read:* takes a shared register r and simply returns its value.
- *Write:* takes a shared register r and a value *val*. The value *val* is assigned to r.
- *Fetch-and-store* (FAS): takes a shared register r and a local register ℓ, and atomically assigns the value of ℓ to r and returns the previous value of r. (The fetch-and-store operation is also called *swap* in the literature.)

Table 1 Comparing the properties of our algorithm with those of several GME algorithms.

GME Algorithms	Group bounded exit BE/GBE	Group concurrent entering CE/GCE	Fairness FCFS/ FIFE	Unknown number of processes & sessions	Shared space for all processes	RMR in CC	RMR in DSM	Hardware used
Joung 1988	BE	CE	✗	✗	$O(n)$	∞	∞	read/write
Keane & Moir 1999	✗	✗	✗	✗	$O(n)$	$O(n)$	$O(n)$	read/write
Hadzilacos 2001	BE	CE	FCFS	✗	$O(n^2)$	$O(n^2)$	∞	read/write
Jayanti et.al. 2003	BE	CE	FCFS FIFE	✗	$O(n)$	$O(n^2)$	∞	read/write
Danek&Hadzilacos 2004	BE	CE	FCFS FIFE	✗	$O(n^2)$	$O(n)$	$O(n)$	CAS fetch&add
Bhatt & Huang 2010	BE	CE	✗	✗	$O(mn)$	$O(min(k, \log n))$	∞	LL/SC
He et. al. 2018	BE	CE	FCFS	✗	$O(n)$	$O(n)$	∞	read/write
Aravid&Hesselink 2019	BE	✗	FIFE	✗	$O(L)$	$O(1)$	∞	fetch&inc
Gokhale & Mittal 2019	BE	CE	✗	✗	$O(n^2)$	$O(min(c, n))$	$O(n)$	CAS fetch&add
Our algorithm	GBE	GCE	FCFS FIFE	✓	$O(n)$	$O(1)$	$O(1)$	CAS fetch&store

✓ - satisfies the property	k - point contention	BE - bounded exit
✗ - does not satisfy the property	c - interval contention	GBE - group bounded exit
n - number of processes	L - a constant number	CE - concurrent entring
m - number of sessions	$s.t.\ L > min(n, m)$	GCE - group concurrent entring

- *Compare-and-swap* (CAS): takes a shared register r, and two values: *new* and *old*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise, r is left unchanged and the value *false* is returned.

Most modern processor architectures support the above operations.

2.2 The CC and DSM machine architectures

We consider two machine architecture models: (1) Cache coherent (CC) systems, where each process (or processor) has its own private cache. When a process accesses a shared memory location, a copy of it migrates to a local cache line and becomes locally accessible until some other process updates this shared memory location and the local copy is invalidated; (2) Distributed shared memory (DSM) systems, where instead of having the "shared memory" in one central location, each process "owns" part of the shared memory and keeps it in its own local memory. These different shared memory models are illustrated in Figure 1.

A shared memory location is locally accessible to some process if it is in the part of the shared memory that physically resides on that process' local memory. Spinning on a remote memory location while its value does not change, is counted only as *one* remote operation that causes communication in the CC model, while it is counted as *many* operations that cause communication in the DSM model. An algorithm satisfies *local spinning* (in the CC or DSM models) if the only type of spinning required is local spinning.

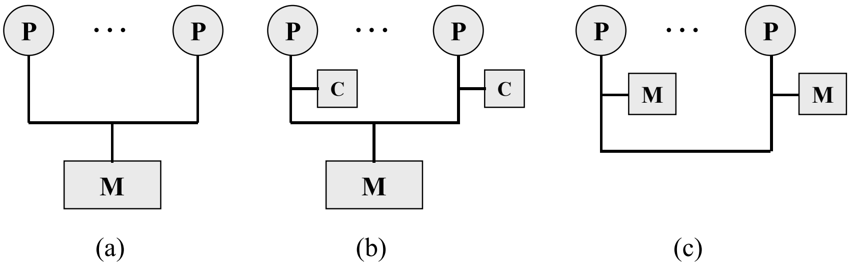


Figure 1 Shared memory models. (a) Central shared memory. (b) Cache Coherent (CC). (c) Distributed Shared Memory (DSM). P denotes processor, C denotes cache, M denotes shared memory.

2.3 RMR complexity: counting remote memory references

We define a *remote reference* by process p as an attempt to reference (access) a memory location that does not physically reside in p's local memory or cache. The remote memory location can either reside in a central shared memory or in some other process' memory.

Next, we define when remote reference causes *communication*. (1) In the DSM model, any remote reference causes communication; (2) in the CC model, a remote reference to register r causes communication if (the value of) r is not (the same as the value) in the cache. That is, communication is caused only by a remote write access that overwrites a different value or by the first remote read access by a process that detects a value written by a different process.

Finally, we define time complexity when counting only remote memory references. This complexity measure, called RMR complexity, is defined with respect to either the DSM model or the CC model, and whenever it is used, we will say explicitly which model is assumed.

- *The RMR complexity* in the CC model (resp. DSM model) is the maximum number of remote memory references which cause communication in the CC model (resp. DSM model) that a process, say p, may need to perform in its entry and exit sections in order to enter and exit its critical section since the last time p started executing the code of its entry section.

3 The GME Algorithm

Our algorithm has the following properties: (1) it has constant RMR complexity in both the CC and the DSM models, (2) it does not require to assume that the number of participating processes or the number of sessions is a priori known, (3) it uses constant space per process, (4) it satisfies FCFS and FIFE fairness, (5) it satisfies the properties: mutual exclusion, starvation-freedom, SGCE, and GBE, (6) it uses an atomic instruction set supported by most modern processor architectures (i.e., read, write, FAS and CAS).

3.1 An informal description

The algorithm maintains a queue of nodes which is implemented as a linked list with two shared objects, *Head* and *Tail*, that point to the first and the last nodes, respectively. Each node represents a request of a process to attend a specific session. A node is an object with a pointer field called *next*, a boolean field called *go*, an integer field called *session*, and two status fields called *status* and *active*. Each process p has its own *two* nodes, called $Nodes_p[0]$ and $Nodes_p[1]$, which can be assumed to be stored in the process p's local memory in a DSM machine, and in the shared memory in a CC machine. Each time p wants to enter its CS section, p uses alternately one of its two nodes. We say that a process p is *enabled* if p can enter its CS in a bounded number of its own steps.

In its doorway, process p initializes the fields of its node as follows:

- *session* is set to the session p wants to attend, letting other processes know the session p is requesting (line 2).
- *next* is a pointer to the successor's node and is initially set to null. This field is being updated later by p's successor (line 11).
- *go* is set to *false*. Later, if p is not enabled, p would spin on its *go* bit until the value is changed to *true*. The *go* bit is the only memory location a process may spin on.
- *status* is set to *WAIT*. This field is being used to determine if a process is enabled. When a process becomes enabled, it sets this field to *ENABLED* (line 26). When process p sees that its predecessor is not enabled (line 13), p spins on its *go* bit (line 14). Otherwise, p informs its predecessor that p has seen that the predecessor is enabled (and hence p does not need help), by setting its predecessor's *status* field to NO_HELP. When a process p sees that its *status* is *ENABLED* (line 30), p tries to help its successor to become enabled and notifies the successor by setting p's own *status* to TRY_HELP.
- *active* is set to *YES*. This field is being used to determine whether p's node is active or not. A node is active if there is a process p that is currently using the node in an attempt to enter p's critical section.

At the end of its doorway, process p threads its node to the end of the queue (line 7). Afterward, p checks what its state is. The state can be one of the following:

1. its node is the first in the queue,
2. its predecessor requests the same session, or
3. its predecessor requests a different session.

In the first case, p can safely become enabled and enters its CS. In the second case, p becomes enabled only if its predecessor is enabled. In the third case, p eventually becomes enabled, once all the processes it follows completed their CSs. We observe that in the exit section, each process causes *Head* to be advanced by exactly one step. So, if p's predecessor's node is inactive, it implies that all the processes that p follows completed their CSs, and thus, p can become enabled and enters its CS.

In the last two cases, once p is enabled, p checks whether it should help its predecessor advance *Head*, by checking if p's predecessor's node is inactive. If the predecessor's node is inactive, then *Head* should point to the node after this inactive node, which is p's node. Therefore, in such a case, p advances *Head* to point to its node.

Once p is enabled to enter its CS, p notifies its successor by setting p's *status* to *ENABLED*. Next, p checks if it has a successor that requests the same session and needs help also to become enabled. If so, p tries to help its successor to become enabled. Only then p enters its CS. The processes that may enter their CS simultaneously are: the process, say p, that *Head* points to its node, and every process q that (1) requests the same session as p, and (2) no conflicting process entered its node between p's node and q's node.

Most of the exit code is wrapped by a mutual exclusion lock. This ensures that each process can cause *Head* to be advanced by a single step every time a process completes its CS. A process that completes its CS and succeeds in acquiring the ME lock tries to advance

Head. If the process succeeds in advancing *Head*, then *Head* value is either *null* or points to the next node in the queue. If *Head* is not *null*, the process changes the *go* bit to *true* in the node that *Head* points to. By doing so, the process lets the next process becoming enabled.

If p fails to advance *Head*, this means that some other process either,

1. enters the queue after p sets *Tail* to *null* (line 38),
2. enters the queue but has not notified its predecessor yet (line 11), or
3. has not entered the queue yet (line 7).

In the first case, the process, say q, in its entry section overrides *Head* to point to q's node (line 9) because q's predecessor is *null*, and so q "advances" *Head* for p. In the latter cases, q in its entry section overrides $Head$ to point to q's own node because it sees q's predecessor's node is inactive, and so q "advances" *Head* for p. Afterward, p releases the ME lock, changes the index of its current node (for the next attempt to enter p's critical section), and completes its exit section.

To guarantee that our GME algorithm satisfies group bounded exit, the mutual exclusion used in the exit section (lines 36 and 49) must satisfy three properties, (1) starvation-freedom, (2) bounded exit, and (3) a property that we call *bounded entry.* Bounded entry is defined as follows: If a process p is in its entry section, while no other process is in its critical section or exit section, some process can complete its entry section within a bounded number of its own steps.[1] While the important and highly influential MCS lock [28] does not satisfy bounded exit, there are variants of it, like the mutual exclusion algorithms from [10, 13, 19], that satisfy all the above three properties.

We will use one of the mutual exclusion algorithms from [13, 19], since (in addition to satisfying the above three properties) each one of these algorithms satisfies the following properties which match those of our GME algorithm: (1) it has constant RMR complexity in both the CC and the DSM models, (2) it does not require to assume that the number of participating processes is a priori known, (3) it uses constant space per process, (4) it satisfies FCFS, (5) it uses the same atomic instruction set as our algorithm, (6) it makes no assumptions on what and how memory is allocated (in [10] it is assumed that all allocated pointers must point to even addresses).

3.2 The algorithm

Two memory records (nodes) are allocated for each process. On DSM machines, these two records reside in the process local memory; on CC machines, these two records reside in the shared memory. In the algorithm, the following symbols are used:

- **&** – this symbol is used to obtain an object's memory location address (and not the value in this address). For example, $\&var$ is the memory location address of variable var.
- **→** – this symbol is used to indicate a pointer to data of a field in a specific memory location. For example, assume *var* is a variable that is a struct with a field called $number$. We now define another variable $loc := \&var$ s.t. loc points to var. Using $loc \rightarrow number$ we would get the value of $var.number$.
- **Q** – the queue in the algorithm is denoted by Q. Q is only used for explanations and does not appear in the algorithm's code.

[1] It is interesting to notice that the bounded entry property cannot be satisfied by a ME algorithm that uses only read/write atomic registers [1], [30] (page 119).

Algorithm 1 The GME algorithm: Code for process p.

```
Type:    QNode: { session: int, go: bool, next: QNode*,
                  active: ∈ {YES, NO, HELP}
                  status: ∈ {ENABLED, WAIT, TRY_HELP, NO_HELP} }
Shared:  Head: type QNode*, initially null                        ▷ pointer to the first node in Q
         Tail: type QNode*, initially null                        ▷ pointer to the last node in Q
         Lock: type ME lock                                       ▷ mutual exclusion lock
         $Nodes_p[0,1]$: each of type QNode, initial value immaterial   ▷ nodes local to $p$ in DSM
Local:   $s$: int                                                 ▷ the session of $p$
         $node_p$: type QNode*, initial value immaterial          ▷ pointer to $p$'s currently used node
         $pred_p$: type QNode*, initial value immaterial          ▷ pointer to $p$'s predecessor node
         $next_p$: type QNode*, initial value immaterial          ▷ pointer to $p$'s successor node
         $temp\_head_p$: type QNode*, initial value immaterial    ▷ temporarily save the head
         $current_p$: ∈ {0, 1}, initial value immaterial          ▷ the index for $p$'s current node

procedure THREAD($s$: int)                                        ▷ $s$ is the session $p$ wants to attend
                                                                  ▷ Begin Doorway
    $node_p := \&Nodes_p[current_p]$        ▷ pointer to current node for this attempt to enter $p$'s CS
    $node_p \to session := s$                                     ▷ $p$'s current session
    $node_p \to go :=$ false                                      ▷ may spin locally on it later
    $node_p \to next :=$ null                                     ▷ pointer to successor
    $node_p \to status := WAIT$                                   ▷ $p$ isn't enabled
    $node_p \to active := YES$                                    ▷ $p$'s node is active

    $pred_p :=$ FAS(Tail, $node_p$)                               ▷ $p$ enters its current node to Q
                                                                  ▷ End Doorway
    if $pred_p =$ null then                                       ▷ was Q empty before $p$ entered?
        Head := $node_p$                                          ▷ $node_p$ is the first in Q
    else                                                          ▷ $p$ has pred
        $pred_p \to next := node_p$                               ▷ notify pred
        if $pred_p \to session = s$ then                          ▷ do we have the same session?
            if not CAS($pred_p \to status$, $ENABLED$, $NO\_HELP$) then
                                                                  ▷ should wait for help from pred?
                await $node_p \to go =$ true          ▷ wait until released by pred with the same session
            else if not CAS($pred_p \to active$, $YES$, $HELP$) then    ▷ should help advance Head?
                Head := $node_p$                                  ▷ help advance Head
            end if
        else                                                      ▷ we have different sessions
            if CAS($pred_p \to active$, $YES$, $HELP$) then       ▷ pred's node is still active?
                await $node_p \to go =$ true
                                       ▷ wait until release by a process with a different session
            else                                   ▷ pred's node is inactive in Q thus $p$ is enabled
                Head := $node_p$
            end if
        end if
    end if

    $node_p \to status := ENABLED$                                ▷ can enter the CS
                                                                  ▷ Try helping the successor
    $next_p := node_p \to next$                                   ▷ save next pointer locally
    if $next_p \neq$ null then                                    ▷ has successor?
        if $next_p \to session = s$ then                          ▷ we have the same session
            if CAS($node_p \to status$, $ENABLED$, $TRY\_HELP$) then
```

```
31:             $next_p \rightarrow go$ := true                    ▷ make your successor enabled
32:         end if
33:     end if
34: end if

35: critical section

36: $Acquire(Lock)$                                                ▷ Mutual exclusion entry section

37: $temp\_head_p$ := Head                                         ▷ save current head locally
38: if CAS(Tail, $temp\_head_p$, null) then     ▷ remove node from tail if it is the only node in Q
39:     CAS(Head, $temp\_head_p$, null)                            ▷ try removing it from the head
40: else if $temp\_head_p \rightarrow next \neq$ null then         ▷ head has successor
41:     $temp\_head_p := temp\_head_p \rightarrow next$            ▷ advance the temp head
42:     Head := $temp\_head_p$                                     ▷ advance the head
43:     $temp\_head_p \rightarrow go$ := true                      ▷ enable the new head
44: else if not CAS($temp\_head_p \rightarrow active$, $YES$, $NO$) then
                       ▷ someone in Tail but hasn't notified to its predecessor in time
45:     $temp\_head_p := temp\_head_p \rightarrow next$            ▷ advance the temp head
46:     Head := $temp\_head_p$                                     ▷ advance the head
47:     $temp\_head_p \rightarrow go$ := true                      ▷ enable the new head
48: end if

49: $Release(Lock)$                                                ▷ Mutual exclusion exit section

50: $current_p := 1 - current_p$                                   ▷ toggle for further use
end procedure
```

3.3 Further explanations

To better understand the algorithm, we explain below several delicate design issues which are crucial for the correctness of the algorithm.

1. *Why does each process p need two nodes $Nodes_p[0]$ and $Nodes_p[1]$?* This is done to avoid a *deadlock*. Assume each process has a single node instead of two, and consider the following execution. Suppose p is in its CS, and q completed its doorway. p resumes and executes its exit section. p completes its exit section while q is in the queue but has not notified p that q is p's successor (line 11). p leaves its *status* field as *ENABLED* and changes its *active* field to *NO* (line 44), so q should be able to enter its CS, no matter what session q requests. p starts another attempt to enter its CS, before q resumes and executes either line 13 or line 19 (depends on which session p requests). p uses its single node and sets *status* to *WAIT* and *active* to *YES* in its doorway (lines 5 and 6, respectively). Now, q continues and (by executing either line 13 or line 19) sees that p is not enabled and p's node is active, so q spins on its *go* bit. Also, p (by executing either line 13 or line 19) sees that q is not enabled and its node is active, so p also spins on its *go* bit. No process will release q, and a deadlock occurs. This problem is resolved by having each process owns two nodes.
2. *Why do we need the CAS operations at lines 13 and 30?* The CAS operations at these lines prevent a potential race condition that may violate the *mutual exclusion* property. Assume we replace the CAS operations at lines 13 and 30, as follows:

- At line 13, p checks if $pred_p \rightarrow status \neq ENABLED$. If so, p waits at line 14. Otherwise, at line **14.5**, p executes $pred_p \rightarrow status = NO\_HELP$.
- At line 30, p checks if $node_p \rightarrow status = ENABLED$. If so, at line **30.5**, p executes $node_p \rightarrow status = TRY\_HELP$ and then continues to line 31 and helps p's successor.

Suppose p is the predecessor of q, and they both request the same session s. p executes line 30, sees that p's *status* is *ENABLED*, and continues to line 30.5 but does not execute this statement yet. Then, q executes line 13, sees that p's *status* is *ENABLED*, executes line 14.5, changes p's *status* to NO_HELP and continues to q's CS. q completes its CS, executes q's exit section, and starts the algorithm again using q's second node. q requests the same session as before, s, and continues to q's CS since q's predecessor is enabled. q completes its exit code and enters the entry code again using q's first node, but now q requests a different session $s' \neq s$. Notice, q's first node is the same node that p has seen as its successor. q continues to line 20 (because it does not request the same session as its predecessor). And so, q waits until its *go* bit is set to *true*. Now, p executes line 30.5 that changes p's *status* to TRY_HELP, continues to line 31 that sets q's first node's *go* bit to *true* and enters its CS. q sees that its *go* bit is *true* and also enters its CS. Therefore, both p and q, which request different sessions, are in their CSs at the same time.

3. *Why do we need the CAS operations at lines 15, 19, and 44?* The CAS operations at these lines are used to prevent a potential race condition that may cause a *deadlock*. Assume we replace the CAS operations are at lines 15, 19, and 44, as follow:

   - At line 15, p checks if $pred_p \rightarrow active \neq YES$. If so, p sets *Head* to its node at line 16. Otherwise, at line **16.5**, p executes $pred_p \rightarrow active = HELP$.
   - At line 19, p checks if $pred_p \rightarrow active = YES$. If so, at line **19.5**, p executes $pred_p \rightarrow active = HELP$.
   - At line 44, p checks if $temp\_head_p \rightarrow active = YES$. If so, p executes lines 45-47 and advances *Head*. Otherwise, p continues to line **47.5** and executes $temp\_head_p \rightarrow active = NO$.

   Suppose p is at line 44 while its successor q is at line 15. q executes line 15 and sees its predecessor's node's *active* equals to *YES*. So q continues to line 16.5 but does not execute it yet. Now, p continues and sees that the *active* field of the first node in the queue is *YES*, so p continues to line 47.5. Then, p sets this node's *active* field to *NO*, while q sets it to *HELP*. Next, p completes its exit section and q enters its CS. Since no process advanced *Head*, *Head* still points to the same node. Assume another process, r, wants to enter its CS and requests a different session than q. r starts the algorithm and gets q's node as its predecessor's node (at line 7). r continues to line 19, as r requests a different session than its predecessor q, and sees that its predecessor's node's *active* field is set to *YES*. Then, r continues to line 19.5, notifying that it did not help to advance *Head*, and waits at line 20 for the *go* bit to be set to *true*. q completes its CS, advances *Head* at line 42, sets the new first node's *go* bit to *true* (line 43), and completes its exit code. But the new first node is q's node, since no process advanced *Head* when p completed its CS. All the new processes will wait until r becomes enabled, but no process can help r becoming enabled and a deadlock occurs.

4. *Why don't we use a dummy node?* The head is being set for the first time at line 9 by the first process that executes the algorithm. The head can be set in line 9 only by one process, the first process, because of the use of the FAS operation at line 7. Only the first process returns null from this operation. The other times that a process may set the head at line 9 is when another process, say q, sets the tail to null in q's exit section, and then q should set the head to null and clear the queue. That means the algorithm is returned to its initial state.

5. *Why have we added line 39, although the algorithm is correct without line 39?* We can remove line 39, and the algorithm would still be correct, as we would override Head at line 9 with the next process that executes the algorithm. We have added this line for semantics reasons, as we do not want to get into a situation where Head points to a node that is no longer active while there are no processes that want to execute the algorithm. That is, when no processes are executing the algorithm, Head and Tail should be null.
6. *Is it essential to include lines 43 and 47 within the ME critical section?* We can move lines 43 and 47 outside the ME critical section (CS), and the algorithm would still be correct. At these lines, we use a local variable *temp_head*, which no other process can change. We placed these lines inside the ME CS for better readability. If we move these lines outside the ME CS, we would need to check if we executed line 38, line 40, or line 44, and only if we executed lines 40 or 44, we then should set *go*.
7. *Who can set process p's go bit to true when p waits at line 20?* By inspecting the code, we can see that p's *go* bit can be changed to *true* either in the entry section (line 31) or in the exit section (lines 43 and 47). Assume p spins on its *go* bit at line 20. p would stop spinning when its *go* bit changed to *true* by another process. Since p is at line 20, p has already tested the condition at line 12 and got *false*. This means that p has requested a different session than its predecessor. Thus, p's predecessor will not reach line 31 because the predecessor will see (line 29) that its successor requests a different session. Each process that acquires the ME lock causes *Head* to be advanced by exactly one step. Therefore, the process that will change p's *go* bit to *true* is the last process that acquires the ME lock and requests the same session as p's predecessor.
8. *The algorithm might become simpler if one can obviate the use of Head. Is the use of Head necessary?* We have tried to simplify the algorithm by not using Head, as done for mutual exclusion in the implementation of the MCS lock [28]. Solving the GME problem is more complex than solving ME. There are more possible race conditions that should be avoided, and using Head helped us in the design of the algorithm. In particular, in the exit code, in lines 43 & 47 the new process at the head of the queue is enabled, by a process that is exiting. We do not see how to implement this in constant time without using Head.

4 Correctness Proof

In this section, we prove that the algorithm satisfies the following properties.

▶ **Theorem 1.** *The GME algorithm has the following properties:*

1. *it satisfies mutual exclusion, starvation-freedom, strong group concurrent entering, and group bounded exit;*
2. *it satisfies FCFS and FIFE fairness;*
3. *it has constant RMR complexity in both the CC and the DSM models;*
4. *it does not require to assume that the number of participating processes or the number of sessions is a priori known;*
5. *it uses constant space per process;*
6. *it uses an atomic instruction set supported by most modern processor architectures, namely, read, write, fetch&store (FAS) and compare&swap (CAS).*

For the lack of space, the proof is omitted. A very detailed proof of Theorem 1 appears in [27].

5 Discussion

With the wide availability of multi-core systems, synchronization algorithms like GME are becoming more important for programming such systems. In concurrent programming, processes (or threads) are often sharing data structures and databases. The GME problem deals with coordinating access to such shared data structures and shared databases.

We have presented a new GME algorithm that is the first to satisfy several desired properties. Based on our algorithm, it would be interesting to design other GME algorithms, such as abortable GME [18] and recoverable GME [15], which will preserve the properties of our algorithm.

References

1 R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, 1992.

2 A. Aravind and W.H. Hesselink. Group mutual exclusion by fetch-and-increment. *ACM Trans. Parallel Comput.*, 5(4), 2019.

3 R. Atreya, N. Mittal, and S. Peri. A quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set. *IEEE Transactions on Parallel and Distributed Systems*, 18(10), 2007.

4 J. Beauquier, S. Cantarell, A. K. Datta, and F. Petit. Group mutual exclusion in tree networks. In *Proc. of the 9th International Conference on Parallel and Distributed Systems*, pages 111–116, 2002.

5 V. Bhatt and C.C. Huang. Group mutual exclusion in $O(log\ n)$ RMR. In *Proc. 29th ACM Symp. on Principles of Distributed Computing*, pages 45–54, 2010.

6 G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronization. In *Proc. of the 13th Annual Symposium on Parallel Algorithms and Architectures*, pages 122–133, 2001.

7 I. Calciu, D. Dice, Y. Lev, V. Luchangco, V.J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM symposium on Principles and practice of parallel programming*, PPoPP '13, page 157–166, February 2013.

8 K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646, 1984.

9 P.L. Courtois, F. Heyman, and D.L Parnas. Concurrent control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, 1971.

10 T.S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR-93-02-02, Dept. of Computer Science, Univ. of Washington, 1993.

11 R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *18th international symposium on distributed computing*, 2004. *LNCS 3274* Springer Verlag 2004, 71–85.

12 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

13 R. Dvir and G. Taubenfeld. Mutual exclusion algorithms with constant rmr complexity and wait-free exit code. In *Proc. of the 21st international conference on principles of distributed systems (OPODIS 2017)*, October 2017.

14 S. Gokhale and N. Mittal. Fast and scalable group mutual exclusion, 2019. `arXiv:1805.04819`.

15 W. Golab and A. Ramaraju. Recoverable mutual exclusion. In *Proc. 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.

16 V. Hadzilacos. A note on group mutual exclusion. In *Proc. 20th symp. on Principles of distributed computing*, pages 100–106, 2001.

17 Y. He, K. Gopalakrishnan, and E. Gafni. Group mutual exclusion in linear time and space. *Theoretical Computer Science*, 709:31–47, 2018.

18 P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, pages 295–304, 2003.

19 P. Jayanti, S. Jayanti, and S. Jayanti. Towards an ideal queue lock. In *Proc. 21st International Conference on Distributed Computing and Networking*, ICDCN 2020, pages 1–10, 2020.

20 P. Jayanti, S. Petrovic, and K. Tan. Fair group mutual exclusion. In *Proc. 22th ACM Symp. on Principles of Distributed Computing*, pages 275–284, July 2003.

21 Yuh-Jzer Joung. Asynchronous group mutual exclusion. In *Proc. 17th ACM Symp. on Principles of Distributed Computing*, pages 51–60, 1998.

22 Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

23 H. Kakugawa, S. Kamei, and T. Masuzawa. A token-based distributed group mutual exclusion algorithm with quorums. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1153–1166, 2008.

24 P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proc. 18th ACM Symp. on Principles of Distributed Computing*, pages 23–32, 1999.

25 P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7), 2001.

26 L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

27 L. Maor. Constant RMR group mutual exclusion for arbitrarily many processes and sessions. Master's thesis, The Interdisciplinary Center, Herzliya, Israel, August 2021.

28 J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

29 M. Takamura, T. Altman, and Y. Igarashi. Speedup of Vidyasankar's algorithm for the group k-exclusion problem. *Inf. Process. Lett.*, 91(2):85–91, 2004.

30 G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.

31 M. Toyomura, S. Kamei, and H. Kakugawa. A quorum-based distributed algorithm for group mutual exclusion. In *Proc. of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 742–746, 2003.

32 K. Vidyasankar. A simple group mutual ℓ-exclusion algorithm. *Inf. Process. Lett.*, 85(2):79–85, 2003.

33 Kuen-Pin Wu and Yuh-Jzer Joung. Asynchronous group mutual exclusion in ring networks. In *Proc. 13th Inter. Parallel Processing Symposium and 10th Symp. on Parallel and Distributed Processing*, pages 539–543, 1999.

Efficient CONGEST Algorithms for the Lovász Local Lemma

Yannic Maus
Technion, Haifa, Israel

Jara Uitto
Aalto University, Espoo, Finland

Abstract

We present a poly log log n time randomized CONGEST algorithm for a natural class of Lovász Local Lemma (LLL) instances on constant degree graphs. This implies, among other things, that there are no LCL problems with randomized complexity between $\Omega(\log n)$ and poly log log n. Furthermore, we provide extensions to the network decomposition algorithms given in the recent breakthrough by Rozhoň and Ghaffari [STOC2020] and the follow up by Ghaffari, Grunau, and Rozhoň [SODA2021]. In particular, we show how to obtain a large distance separated weak network decomposition with a negligible dependency on the range of unique identifiers.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, CONGEST, Lovász Local Lemma, locally checkable labelings

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.31

Related Version *Full Version*: https://arxiv.org/abs/2108.02638

Funding This project was partially supported by the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 755839 (Yannic Maus).

1 Introduction

Our main contribution is a poly log log n round randomized distributed CONGEST algorithm to solve a natural class of Lovász Local Lemma (LLL) instances on constant degree graphs. Among several other applications, e.g., various defective graph coloring variants, this implies that there is no LCL problem (locally checkable labeling problem) with randomized complexity strictly between poly log log n and $\Omega(\log n)$, which together with the results of [3] implies that the *known world* of complexity classes in the sublogarithmic regime in the CONGEST and LOCAL model are almost identical. As a side effect of our techniques we extend the understanding of the computation of network decompositions in the CONGEST model. We now explain our contributions on LLL and LCLs in more detail; in the second half of the introduction we explain our techniques, contributions on network decomposition and how they relate to results in the LOCAL model.

We work in the CONGEST model of distributed computing, where a network of computational devices is abstracted as an n-node graph, where each node corresponds to a computational unit. In each synchronous round, the nodes can send messages of size $b = O(\log n)$ to their neighbors. In the end of the computation, each node is responsible of outputting its own part of the output/solution, e.g., its color in a graph coloring problem. The LOCAL model is otherwise the same, except that there is no bound on the message size b.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 31; pp. 31:1–31:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Background on LLL and LCLs in LOCAL

An *instance* of the *Lovász Local Lemma problem* is formed by a set of variables and a set of bad event $\mathcal{E}_1, \ldots, \mathcal{E}_n$ that depend on the variables. The famous Lovász Local Lemma [23] states that if the probability of each event is upper bounded by p, each event only shares variables with at most d other events and $epd < 1$ holds, then there exists an assignment to the variables that avoids all bad events. An example of a problem that can be solved via LLL is the *sinkless orientation* problem on graphs with minimum degree 4. In the sinkless orientation problem the objective is to orient the edges of a graph such that every vertex has at least one outgoing edge. This can be modeled by an LLL as follows: Orient each edge randomly – each edge represents a variable – and introduce a bad event for each vertex that holds if and only if all edges are oriented towards it. One obtains $p = 2^{-d}$ and $d \geq 4$ such that $epd < 1$ holds. The Lovász Local Lemma has had a huge success in theory of computation. One highlight is the beautiful and simple parallel algorithm by Moser and Tardos [39]. In the distributed version of the problem each variable and each event is simulated by one node of the communication network such that the variables of each event $\mathcal{E}$ are simulated by the node simulating $\mathcal{E}$ or one of its neighbors. The algorithm by Moser and Tardos immediately yields a randomized $O(\log^2 n)$ LLL algorithm in the LOCAL model. Often one has stronger guarantees on the relation of p and d, e.g., a polynomial criterion $epd^2 < 1$ (or even larger exponents) instead of only $epd < 1$, and can obtain simpler and faster algorithms. E.g., in the same model, Chung, Pettie, and Su obtained a randomized algorithm that runs in time $O(\log_{epd^2} n)$ whenever the *criterion* $epd^2 < 1$ holds [20].
LCLs through LLL: The main recent interest in distributed LLL are constant degree graphs motivated by the study of LCLs. LCL problems are defined on constant degree graphs and in an *LCL problem* each node is given an input from a constant sized set of labels and must output a label from a constant sized set of labels. The problem is characterized by a set of feasible constant-radius neighborhoods (for a formal definition see Section 4). Many classic problems are LCL problems, e.g., the problem of finding a vertex coloring with Δ colors in a graph with maximum degree Δ. The systematic study of LCLs in the LOCAL model initiated by Naor & Stockmeyer [40] has picked up speed over the last years leading to an almost complete classification of the complexity landscape of LCL problems in the LOCAL model [18, 19, 4, 2, 43]. One of the most important results in this line of research is by Chang and Pettie [19] who showed that any $o(\log n)$-round randomized LOCAL algorithm for an LCL problem P implies the existence of a T_{LLL}-round algorithm for P where T_{LLL} is the runtime of an LLL algorithm with an (arbitrary) polynomial criterion, e.g., for $p(ed)^{100}$. Among other implications, the breakthrough result by Rozhoň and Ghaffari provided an $(\operatorname{poly}\log\log n)$-round LLL algorithm, in the LOCAL model, for the case that the dependency degree d is bounded. This implies a gap in the randomized complexity landscape for LCLs. There is no LCL problem with a complexity strictly between $\operatorname{poly}\log\log n$ and $\Omega(\log n)$ in the LOCAL model.

1.1 Our Results on the Distributed Lovász Local Lemma and LCLs

Motivated by the progress in LOCAL using the *LCLs through LLL* application, we aim to design CONGEST algorithms for the LLL problem. The first observation is that one wants to limit the range of the variables, as any reasonable algorithm should be able to send the value of a variable in a CONGEST message, and one also wants to limit the number of variables at a node such that nodes can learn assignments of all variables associated with their bad event(s) efficiently. We call an LLL instance with dependency degree *d range bounded* if each event only depends on $\operatorname{poly} d$ variables and each variable uses values from a range of size $\operatorname{poly} d$. At first sight, this seems like a a very restrictive setting, but in fact most instances of LLL satisfy these requirements. Our main result is contained in the following theorem and meets the $\operatorname{poly}\log\log n$ round state of the art in the LOCAL model [43].

▶ **Theorem 1.** *There is a randomized* CONGEST *algorithm that with high probability solves any range bounded Lovász Local Lemma instance that has at most n bad events, dependency degree $d = O(1)$ and satisfies the LLL criterion $p(ed)^8 < 1$ where p is an upper bound on the probability that a bad event occurs, in* $\operatorname{poly}\log\log n$ *rounds.*

On the negative side, it is know that a double logarithmic dependency cannot be avoided. There is an $\Omega(\log\log n)$-round lower bound for solving LLL instances with randomized algorithms in the LOCAL model [11], that holds even for constant degree graphs, and it naturally applies to the CONGEST model as well. It stems from a lower bound on the aforementioned sinkless orientation problem.

Implications for the Theory of LCLs

As the reduction from [19], that reduces sublogarithmic time randomized algorithms to LLL algorithms, immediately works in CONGEST, our fast LLL algorithm implies a gap in the complexity landscape of LCLs. A more precise definition of LCLs and a proof for the following corollary will be presented in the end of Section 4.

▶ **Corollary 2.** *There is no LCL problem with randomized complexity strictly between* $\operatorname{poly}\log\log n$ *and* $\Omega(\log n)$ *in the* CONGEST *model.*

In fact, Corollary 2 together with the results of [3] implies that the *known world* of complexity classes in the sublogarithmic regime in the CONGEST and LOCAL model are almost identical. In fact, a difference can only appear in the extremely small complexity regime between $\Omega(\log\log^* n)$ and $O(\log^* n)$ and in the important regime of complexities that lie between $\Omega(\log\log n)$ and $\operatorname{poly}\log\log n$ where complexity classes are not understood in the LOCAL model. An immediate implication of Corollary 2 is a $\operatorname{poly}\log\log n$-round randomized CONGEST algorithm for Δ-coloring when Δ is constant, a result that was previously only known in the LOCAL model [29]. Chang and Pettie [19] conjecture that the runtime of LLL in the LOCAL model is $O(\log\log n)$ on general bounded degree graphs which would further simplify the complexity landscape for LCLs. On trees this complexity can be obtained [17] in the LOCAL model and for the specific LLL instances that arise in the study of LCLs on trees an $O(\log\log n)$-round algorithm has also been found in the CONGEST model [3].

1.2 Technical Overview and Background on our Methods

Our core technical contribution to obtain Theorem 1 is a bandwidth efficient derandomization of the LLL algorithm by Chung, Pettie, and Su [20] that we combine with the shattering framework of Fischer and Ghaffari [24]. To explain these ingredients and how we slightly advance our understanding of network decompositions in the CONGEST model on the way, we begin with explaining the background on distributed derandomization and network decompositions and the relation to results in the LOCAL model.

Background on Network Decompositions and Distributed Derandomization

Network decompositions are a powerful tool with a range of applications in the area of distributed graph algorithms and were introduced by Awerbuch, Luby, Goldberg, and Plotkin [1]. A (C, D)-*network decomposition* (ND) is a partition of the vertices of a graph into C collections of clusters (or color classes of clusters) such that each cluster has diameter[1] at most D. Further, it is required that the clusters in the same collection are *independent*, i.e., are not connected by an edge. This is extremely helpful in the LOCAL model, e.g., to compute a $(\Delta + 1)$-coloring of the network graph one can iterate through the C collections, and within each collection process all clusters in parallel (due to their independence). Due the unbounded message size dealing with a single cluster is trivial. A cluster leader can learn all information about the cluster in time that is proportional to the cluster diameter D, solve the problem locally and disseminate the solution to the vertices of the cluster. Thus the runtime scales as $O(C \cdot D)$. Hence, the objective has been to compute such decompositions as fast as possible and with C and D as small as possible, optimally, all values should be polylogarithmic in n. Awerbuch et al. gave a deterministic LOCAL algorithm with round complexity and C and D equal to $2^{O(\sqrt{\log n \log\log n})} \gg \operatorname{poly}\log n$. Panconesi and Srinivasan improved both parameters and the runtime to $2^{O(\sqrt{\log n})}$ [41]. Linial and Saks showed that the optimal trade-off between diameter and the number of colors is $C = D = O(\log n)$ and they provided an $O(\log^2 n)$-round randomized algorithm to compute such decompositions [38] . These algorithms remained the state of the art for almost three decades even though the need for an efficient deterministic algorithm for the problem has been highlighted in many papers, e.g., [7, 32, 27, 19, 31].

A few years ago, Ghaffari, Kuhn, and Maus [33] and Ghaffari, Harris, and Kuhn [28] highlighted the importance of network decompositions by showing that an efficient deterministic LOCAL algorithm for network decompositions with $C = D = \operatorname{poly}\log n$ would immediately show that an efficient randomized LOCAL algorithm for any efficiently verifiable graph problem would yield an efficient deterministic algorithm. Here, all occurrences of *efficient* mean polylogarithmic in the number of nodes of the network. Then, in the aforementioned breakthrough Rozhoň and Ghaffari [43] devised such an efficient deterministic algorithm for network decompositions, yielding efficient deterministic algorithms for many problems. The result also had an immediate impact on randomized algorithms. Many randomized algorithms in the area use the *shattering technique* that at least goes back to Beck [10] who used the technique to solve LLL instances in centralized settings. It has been introduced to distributed computing by Barenboim, Elkin, Pettie and Schneider in [9]. The shattering technique usually implements the following schematic: First, the nodes use a randomized process and the guarantee is that with high probability *almost* all nodes find a satisfactory output. The remainder of the graph is *shattered* into *small components*, that is, after this so called *pre-shattering* phase the unsolved parts of the graph induce small – think of $N = \operatorname{poly}\log n$ size – connected components. In the *post-shattering phase* one wishes to use an efficient deterministic algorithm, e.g., a deterministic algorithm with complexity $T(n) = \operatorname{poly}\log n$ applied to each small component results in a complexity of $T(N) = T(\operatorname{poly}\log\log n)$. Combining the shattering technique, e.g., [9, 24, 25], the network decomposition algorithm from [43] and the derandomization from [32, 27] the randomized complexities for many graph problems in the LOCAL model are $\operatorname{poly}\log\log n$.

[1] For the sake of this exposition it is enough to assume that a cluster $\mathcal{C}$ has diameter D, that is, any two vertices in the cluster are connected with a path within the cluster of length at most D. Actually, often these short paths are allowed to leave the cluster which may cause congestion when one uses these paths for communication in two clusters in parallel. The details of the standard way to model this congestion and its impact are discussed in Section 3.

The Challenges in the CONGEST Model

The holy grail would be to obtain a similar derandomization result in the CONGEST model. But, even though the network decomposition algorithm from [43] immediately works in CONGEST, we are probably far from obtaining such a result. Even for simple problems like computing a maximal independent set or a $(\Delta+1)$-coloring one has to work much harder to even get $\operatorname{poly}\log n$ round deterministic algorithms in the CONGEST model, even if a $(\log n, \log n)$-network decomposition is given for free [15, 5]. For LLL obtaining such a bandwidth efficient algorithm seems much harder. Even in the LOCAL model one either has to go through the aforementioned derandomization result or one can solve an LLL instance with criterion $p(ed)^{\lambda}$, e.g., think of $\lambda = 10$, if the network decomposition only has λ color classes. This immediately implies [38] that the cluster diameter is $\Omega(n^{1/\lambda})$ and algorithms that rely on aggregating all information about a cluster in a cluster leader must use at least $\Omega(n^{1/\lambda})$ rounds. In the LOCAL model there are black box reductions [32, 43, 8] to compute such decompositions. We provide an analysis of the algorithm by Rozhoň and Ghaffari (and of another algorithm by Ghaffari, Grunau and Rozhoň [26]) where we carefully study the trade-off between number of colors, the diameter of the clusters, and the runtime of the algorithm to obtain such decompositions in CONGEST.

▶ **Theorem 8** (informal). *For $k \geq 1$ and $1 \leq \lambda < \log n$ there is an $O(k \cdot n^{1/\lambda} \operatorname{poly}\log n)$-round deterministic* CONGEST *algorithm to compute a $(\lambda, (k \cdot n^{1/\lambda} \operatorname{poly}\log n)$-network decomposition such that any two clusters with the same color have distance strictly more than k. The algorithm works with a mild dependence on the ID space*[2]*.*

For $k = 1$, decompositions with few colors similar to Theorem 8 could already be obtained (using randomization) through the early works by Linial and Saks, who also showed that their trade-off of the number of colors and the cluster diameter in Theorem 8 is nearly optimal [38]. In fact, Theorem 8 does not just provide an improved analysis of the algorithm of [43] but it also comes with a mild dependence on the identifier space and an arbitrary parameter to increase the distance between clusters. Both ingredients are also present in our second result on network decompositions (Theorem 6) where they are crucial for the proof of Theorem 1.

Our Solution: Range bounded LLLs in CONGEST. We use the shattering framework for LLLs of [24] whose pre-shattering phase, as we show, works in the CONGEST model for range bounded LLLs. As a result we obtain small remaining components of size $N \ll n$, in fact, we obtain $N = O(\log n)$. Furthermore, one can show that the remaining problem that we need to solve on the small components is also an LLL problem. We solve these small instances via a bandwidth efficient derandomization of the LLL algorithm by Chung, Pettie, Su [20]. Note that we cannot solve the small components without derandomizing their algorithm, as running their randomized algorithm on each component for $T(N) = O(\log N) = O(\log\log n)$ rounds would imply an error probability of $1/N$ which is exponentially larger than the desired high probability guarantee of $1/n$. Thus, we desire to find *good random bits* for all nodes with which we can execute the algorithm from [20] without any error. The goal is to use a network decomposition algorithm to partition the small components into $O(\log N) = O(\log\log n)$ collections of independent clusters. Then, we iterate through the collections and want to obtain good random bits for the vertices inside each cluster. Since the randomized runtime of [20] on a small component would be $T(N) = O(\log N) = O(\log\log n)$, we observe that the

[2] To be precise, the dependency on an ID space $\mathcal{S}$ space is a $\log^* |\mathcal{S}|$ factor.

random bits of a node v cannot influence the correctness at a node u if u and v are much further than $T(N)$ hops apart. Thus, similar to Theorem 8 we devise the following theorem to compute network decompositions with large distances between the clusters. In fact, if we apply the theorem with $k > 2 \cdot T(N)$, we obtain independent clusters in each color class of the decomposition.

▶ **Theorem 6** (informal). *For $k \geq 1$ there is an $O(k \cdot \text{poly} \log n)$-round deterministic* CONGEST *algorithm to compute a $(\log n, \text{poly} \log n)$-network decomposition such that any two clusters with the same color have distance strictly more than k. The algorithm works with a mild dependence on the ID space.*

While [43] provided a modification of their algorithm that provides a larger distance between clusters its runtime and cluster diameter depend polylogarithmically not only on the number of nodes in the network but also on the ID space. In [26] Ghaffari, Grunau and Rozhoň have reduced the ID space in the special case in which the cluster distance k equals one. However, our bandwidth efficient LLL algorithm requires $k \gg 1$ and identifier independence at the same time. Thus, one can either say we add the ID space independence to [43] or we extend the results of [26] to $k > 1$.

We already explained why we require that we obtain a network decomposition with a large cluster distance to derandomize an algorithm. The mild dependence on the identifier space in Theorem 6 is also essential as the small components with N nodes on which we want to use the network decomposition algorithm, actually live in the original communication network G. Thus they are equipped with identifiers that are polynomial in n, that is, exponential in N. The fact that the small components live in the large graph makes our life harder when computing a network decomposition but it helps us when designing efficient CONGEST algorithms. The bandwidth when executing an algorithm on the small components is still $O(\log n)$ bits per edge per round while the components are of size $N \ll n$. Ignoring details for the sake of this exposition, we use the increased bandwidth to gather enough information about a cluster in a single cluster leader such that it can select good random bits for all nodes of the cluster and in parallel with all other clusters of the same color class due to the large distance between clusters.

We emphasize that there have been several other approaches to derandomize algorithms in the CONGEST model and discussing all of them here would be well beyond the scope of this work. However, we still believe that our derandomization technique in the post-shattering phase might be of independent interest as it applies to a more general class of algorithms than just the aforementioned LLL algorithm from [20].

Other Implications. As the runtime of Theorem 6 is a $\log n$ factor faster than the result in [43], it also improves the complexity of deterministic distance-2 coloring with $(1+\varepsilon)\Delta^2$ colors in the CONGEST model to $O(\log^7 n)$ rounds when plugged into the framework of [35]. Similar improvements carry over to the approximation of minimum dominating sets [21] and spanner computations [30]. Due to the identifier independence and possibility to increase the distance between clusters Theorem 6 yields an improved randomized complexity of distance-2 coloring. Using Theorem 6 in the shattering based algorithm from [36] improves the runtime for $\Delta^2 + 1$ colors from $2^{O(\sqrt{\log \log n})}$ to $O(\log^7 \log n)$ rounds.

Roadmap

In Section 1.3, we provide pointers for further related work, mainly on network decompositions. In Section 2 we define the models and introduce notation. In Section 3 we formally state the result on network decompositions and indicate the main changes to prior work. Due to space limitations, the formal proofs of these results appear in the full version of the paper. The main part of the paper, Section 4, deals with proving Theorem 1.

1.3 Further Related Work

We already mentioned that [26] provides an efficient deterministic CONGEST algorithm with a mild dependence on the ID space. More precisely, they provided a $(O(\log n, \log^2 n))$-network decomposition in $O(\log^5 n + \log^4 n \cdot \log^* b)$ rounds, where $\log^* b$ is the dependency on the identifier length b [26]. One drawback that we have ignored until now – and that also applies to all of our results – is that these network decompositions only have so called *weak diameter*, that is, the diameter of a cluster is only guaranteed to be small if it is seen as a subset of the communication network G, that is, the distance between two vertices is measured in G and not only in the subgraph induced by a cluster. In contrast, in so called *strong network decompositions* the diameter in the subgraph of G that is induced by each cluster has to be small. Very recently, at the cost of increasing the polylogarithmic factors in the runtime the results of [26] were extended to obtain strong $(O(\log n, \log^2 n))$-network decompositions [16]. Earlier works by Elkin and Neiman provided an $O(\log^2 n)$ randomized algorithm for computing strong $(O(\log n), O(\log n))$-decompositions [22].

Most previous works in the CONGEST model do not ensure large distances between clusters. Besides the result in [43] that we have already discussed there are two notable exceptions. Ghaffari and Kuhn matched the complexity of the LOCAL model algorithm by Awerbuch et al. by giving a deterministic $k \cdot 2^{O(\sqrt{\log n \log\log n})}$ round algorithm [30]. Later, this was improved to $k \cdot 2^{O(\sqrt{\log n})}$ rounds by Ghaffari and Portmann [34]. In both cases, the decomposition parameters were identical to the runtimes. None of these results is the state of the art anymore, except for the fact that they compute strong network decompositions with large distances between the clusters.

Barenboim, Elkin, and Gavoille provide various algorithms with different trade-offs between the number of colors and the cluster diameter, most notably a randomized algorithm to compute a strong network decomposition with diameter $O(1)$ and $O(n^{\varepsilon})$ colors [8]. A similar result with $O(n^{1/2+\varepsilon})$ colors was obtained by Barenboim in [6].

Brandt, Maus, and Uitto and Brandt, Grunau, and Rozhoň have shown that LLL instances with an exponential LLL criterion, that is, $p2^d < 1$ holds, can be solved in $O(\log^* n)$ rounds on bounded degree graphs [14, 12]. Furthermore, it is known that $\Omega(\log^* n)$ rounds cannot be beaten under any LLL criterion that is a function of the dependency degree d [20]. For an exponential criterion that satisfies $p2^d \geq 1$, it follows from the works of Brandt et al. and Chang, Kopelowitz, and Pettie [18] that there is a lower bound of $\Omega(\log n)$ rounds for deterministic LLL. Hence [14, 12] and [18] show that there is a sharp transition of the distributed complexity of LLL at $2p^d = 1$. Before the result in [43] improved the runtime to $\operatorname{poly}\log\log n$, Ghaffari, Harris, and Kuhn gave the state of the art randomized LLL algorithm in the LOCAL model. On constant degree graphs its runtime was described by a tower function and lies between $\operatorname{poly}\log\log n$ and $2^{O(\sqrt{\log\log n})}$ [27]. We are not aware of any works that explicitly studied LLL in the CONGEST model before our paper.

2 Models, LCLs & Notation

Given a graph $G = (V, E)$ the hop distance in G between two vertices $u, v \in V$ is denoted by $dist_G(u, v)$. For a vertex $v \in V$ and a subset $S \subseteq V$ we define $dist_G(v, S) = \min_{u \in S}\{dist(v, u)\} \in \mathbb{N} \cup \{\infty\}$. For two subsets $S, T \subseteq V$ we define $dist_G(S, T) = \min_{v \in T} dist_G(v, S)$. For an integer n we denote $[n] = \{0, \dots, n-1\}$.

The LOCAL and CONGEST Model of distributed computing [37, 42]. In both models the graph is abstracted as an n-node network $G = (V, E)$ with maximum degree at most Δ. Communication happens in synchronous rounds. Per round, each node can send one message to each of its neighbors. At the end, each node has to know its own part of the output, e.g., its own color. In the LOCAL model there is no bound on the message size and in the CONGEST model messages can contain at most $O(\log n)$ bits. Usually, in both models nodes are equipped with $O(\log n)$ bit IDs (polynomial ID space) and initially, nodes know their own ID or their own color in an input coloring but are unaware of the IDs of their neighbors. Randomized algorithms do not use IDs (but they can create them from space $[n^c]$ for a constant c and with a $1/n^c$ additive increase in the error probability), have a fixed runtime and need to be correct with probability strictly more than $1 - 1/n$ (Monte Carlo algorithm). Actually, algorithms are defined such that a parameter n is provided to them, where n represents an upper bound on the number of nodes of the graph and the algorithm's guarantees have to hold on any graph with at most n nodes. For technical reasons in our gap results, we assume that the number of random bits used by each node is bounded by some finite function $h(n)$. We note that the function $h(n)$ can grow arbitrarily fast and that this assumption is made in previous works as well [18, 3].

▶ **LCL definition** ([40]). An *LCL problem* Π is a tuple $(\Sigma_{\text{in}}, \Sigma_{\text{out}}, F, r)$ satisfying the following.

- Both Σ_{in} and Σ_{out} are constant-size sets of labels,
- the parameter r is an arbitrary constant, called the *checkability radius* of Π,
- F is a finite set of pairs $(H = (V^H, E^H), v)$, where:
  - H is a graph, v is a node of H, and the radius of v in H is at most r;
  - Every pair $(v, e) \in V^H \times E^H$ is labeled with a label in Σ_{in} and a label in Σ_{out}.

In an instance of an LCL problem $\Pi = (\Sigma_{\text{in}}, \Sigma_{\text{out}}, F, r)$ on a graph $G = (V, E)$ each node receives an *input label* from Σ_{in}. An algorithm solves Π if, for each node $v \in V$, the radius-r hop neighborhood of v together with the input labels and computed outputs for the nodes lies in the set of *feasible labelings* F.

3 Graph Decompositions

In this section we state our results on network decompositions and provide the respective definitions. At the end of the section we provide a short overview of the techniques that we use to prove the results. The formal proofs appear in the full version of the paper.

3.1 Cluster Collections and Network Decompositions with Congestion

Classically, one defines a *(weak)* (c, d)*-network decomposition* as a coloring of the vertices of a graph G with c colors such that the connected components of each color class have weak diameter at most d. The *weak diameter* of a subset $\mathcal{C} \subseteq V$ is the maximum distance in G that any two vertices of $\mathcal{C}$ have.

We use the following definition that augments one color class of a network decomposition with a communication backbone for each cluster. A *cluster* $\mathcal{C}$ of a graph is a subset of nodes. A *Steiner tree* is a rooted tree with nodes labeled as *terminal* and *nonterminal*.

▶ **Definition 3** (cluster collection). *A collection of clusters of a graph $G = (V, E)$ consists of subsets of vertices $\mathcal{C}_1, \dots, \mathcal{C}_p \subseteq V$ and has Steiner radius β and unique b-bit cluster identifiers if it comes with associated Steiner subtrees $T_1, \dots, T_p$ of G such that clusters are disjoint, i.e., $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset$ for $i \neq j$ and for each $i \in 1, 2, \dots, p$ we have*

1. *cluster $\mathcal{C}_i$ has a unique b-bit identifier $id_{\mathcal{C}_i}$,*
2. *the terminal nodes of Steiner tree T_i of cluster $\mathcal{C}_i$ are formed by $\mathcal{C}_i$ (T_i might contain nodes $\notin \mathcal{C}_i$ as non terminal nodes),*
3. *the edges of Steiner Tree T_i are oriented towards a* cluster leader *$\ell_{\mathcal{C}_i}$,*
4. *Steiner tree T_i has diameter at most β (Steiner tree diameter).*

The collection has congestion κ *if each edge in E is contained in at most κ Steiner trees. When we compute a cluster collection in the* CONGEST *model, we require that each node of a cluster knows the cluster identifier. Furthermore, we require that for each edge of T_i that is incident to some $v \in V$, node v knows $id_{\mathcal{C}_i}$ and additionally v knows the direction of the edge towards the root $r_{\mathcal{C}_i}$.*

We now define our notion of a network decomposition with a communication backbone.

▶ **Definition 4** (network decomposition with congestion). *Let $k \geq 1$. A (C, β)-*network decomposition *with cluster distance k and* congestion κ *of a graph $G = (V, E)$ is a partition of V into C cluster collections with Steiner radius β and congestion κ such that any two clusters $\mathcal{C} \neq \mathcal{C}'$ in the same cluster collection have distance strictly more than k, i.e., $dist_G(\mathcal{C}, \mathcal{C}') > k$.*

The C cluster collections are also called the C color classes *of the decomposition.*

Instead of a network decomposition with cluster distance k, we often just speak of a network decomposition of G^k. In all cases the Steiner trees are given by vertices and edges in G and not by edges in G^k, which would corresponds to paths in G. The default value for k is 1 which we use whenever we do not specify its value. in this case the definition corresponds with the classic network decomposition.

3.2 Network Decomposition and Ball Carving Algorithms

At the core of our network decomposition algorithm is the following *ball carving* result to form one color class of the decomposition, i.e., to form one cluster collection. The name ball carving stems from the original existential proof for network decompositions [1].

▶ **Lemma 5** (ball carving, $k \geq 1$). *Let $k \geq 1$ be an integer and $x \geq 1$ (both potentially functions of n). Then there is a deterministic distributed* CONGEST *(bandwidth b and b-bit identifiers) algorithm that, given a graph $G = (V, E)$ with at most n nodes and a subset $S \subseteq V$, computes a cluster collection $\mathcal{C}_1, \ldots, \mathcal{C}_p \subseteq S$ with*

- $|\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_p| \geq (1 - 1/x) \cdot |S|$,
- *Steiner radius $\beta = O(k \cdot x \cdot \log^3 n)$ and congestion $\kappa = O(\log n \cdot \min\{k, x \cdot \log^2 n\})$,*
- *the pairwise distance in G between $\mathcal{C}_i$ and $\mathcal{C}_j$ for $1 \leq i \neq j \leq p$ is strictly more than k.*

The runtime is $O(k \cdot x \cdot \log^4 n \cdot \log^* b) + O(k \cdot x^2 \cdot \log^6 n \cdot \min\{k, x \cdot \log^2 n\})$ rounds.

By using the aggregation tools for overlapping Steiner trees of [26] Lemma 5 is by a $\log n$ factor faster than the corresponding result in [43]. The ball carving result immediately imply the following network decomposition results.

▶ **Theorem 6** ($O(\log n)$ colors). *For any (potentially non constant) $k \geq 1$, there is a deterministic* CONGEST *algorithm with bandwidth b that, given a graph G with at most n nodes and unique b-bit IDs from an exponential ID space, computes a (weak) $(O(\log n), O(k \cdot \log^3 n))$-network decomposition with cluster distance k and with congestion $O(\log^2 n \cdot \min\{k, \log^2 n\})$ in $O(k \cdot \log^7 n \cdot \min\{k, \log^2 n\})$ rounds.*

We also analyze a more involved ball carving algorithm of [26] for different parameters in order to obtain decompositions with fewer colors. Then Lemmas 5 and 7 imply the following result for network decompositions with few colors.

▶ **Lemma 7** (faster ball carving, $k = 1$). *Let $x \geq 1$ (potentially a function of n). Then there is a deterministic distributed* CONGEST *(bandwidth b and b-bit identifiers) algorithm that, given a graph $G = (V, E)$ with at most n nodes and a subset $S \subseteq V$, computes a cluster collection $\mathcal{C}_1, \ldots, \mathcal{C}_p \subseteq S$ with*

- $|\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_p| \geq (1 - 1/x) \cdot |S|$,
- *Steiner radius* $\beta = O(x \cdot \log^2 n)$ *and congestion* $\kappa = O(\log n)$,
- *the pairwise distance in* G *between* $\mathcal{C}_i$ *and* $\mathcal{C}_j$ *for* $1 \leq i \neq j \leq p$ *is strictly more than* 1.

The runtime of the algorithm is $O(x^2 \log^4 n)$ *rounds.*

▶ **Theorem 8** (few colors). *For any* $\lambda \leq \log n$ *there is a deterministic* CONGEST *algorithm with bandwidth* b *that, given a graph* G *with at most* n *nodes and unique* b*-bit IDs from an exponential ID space, computes a weak* $(\lambda, n^{1/\lambda} \log^2 n)$*-network decomposition of* G *in* $O(\lambda \cdot n^{2/\lambda} \cdot \log^4 n)$ *rounds with congestion* $\kappa = O(\log n)$.

For any (possibly non constant) $k \geq 1$ *and any* $\lambda \leq \log n$ *there is a deterministic* CONGEST *algorithm that, given a graph* G *with at most* n *nodes and unique IDs from an exponential ID space, computes a weak* $(\lambda, k \cdot n^{1/\lambda} \log^3 n)$*-network decomposition of* G^k *in* $O(k \cdot n^{2/\lambda} \cdot \log^6 n \cdot \min\{k, x \log^2 n\})$ *rounds.*

Theorems 6 and 8 also work with more general ID spaces, as long as the IDs fit into a CONGEST message. To simplify the statements and because exponential IDs are (currently) the main application we do not explicitly state their mild dependence on the size of the ID space $\mathcal{S}$ in theorems. It can be quantified by $O(x \cdot \text{poly}\log n \cdot \log^* |\mathcal{S}|)$ where the polylogarithmic terms are strictly dominated by the ones in the theorems.

Summary of Our Ball Carving Contributions. Both, the ball carving results in [43] and [26] begin with each vertex of the to be partitioned set S (see Lemmas 5 and 7) being its own cluster inheriting its node ID as the cluster ID. Then they implement a distributed *ball growing* approach in which vertices leave their cluster, join other clusters, or become dead, i.e., are disregarded; even whole clusters can dissolve, e.g., if all their vertices join a different cluster. The goal is to ensure that at the end no two alive clusters are neighboring, that is, in distance k, that the fraction of vertices of S that are declared dead is small and that the diameter of the clusters does not become too large. In fact, as vertices can leave clusters one only obtains guarantees on the weak diameter of clusters. To this end the algorithm of [43] iterates through the bits of the cluster IDs and in phase i, vertices change their clusters or become dead such that at the end of the i-th phase no two neighboring clusters have the same i-th bit in their cluster ID. If cluster IDs have b bits, then after the b-th phase each remaining cluster is not connected to any other cluster. Thus, the runtime crucially depends on the size of the cluster IDs. The main change in [26] to remove this ID space dependence is to replace the bits (0 or 1) used in phase i with a coloring of the clusters with two colors, red and blue. A new coloring is computed in every phase and the crucial property of the coloring that ensures progress towards the separation of the clusters is that each connected component of clusters has roughly the same number of red and blue clusters. Thus, intuitively, at the end of a phase red and blue clusters are separated and the sizes of all connected components decrease.

In our algorithms we follow the same high level approach but extend the techniques (for identifier "independence") to work with a larger separation between the clusters, and also for the computation of decompositions with fewer colors. A crucial difficulty is that we cannot quickly disregard all dead vertices and reason that each connected components of clusters in G can be treated independently. It might even be that two clusters are connected by a path with $\leq k$ hops and the path might contain dead vertices. Instead, we consider connected components in G^k and always keep all vertices of G in mind. As two clusters that are adjacent in such a component might not be adjacent in the original graph we ensure that no congestion appears *in between* the clusters. Careful algorithm design and reasoning is needed when computing a balanced coloring of clusters in such components.

To obtain the network decompositions with fewer colors, e.g., with $\lambda = 10$ colors, we analyze our algorithm for a suitable choice of parameters and show that all congestion parameters are not affected by the choice of λ.

4 Distributed Range Bounded LLL and its Implications

The objective of this section is to prove the following theorem.

▶ **Theorem 1.** *There is a randomized* CONGEST *algorithm that with high probability solves any range bounded Lovász Local Lemma instance that has at most n bad events, dependency degree $d = O(1)$ and satisfies the LLL criterion $p(ed)^8 < 1$ where p is an upper bound on the probability that a bad event occurs, in* $\operatorname{poly}\log\log n$ *rounds.*

Distributed Lovász Local Lemma (LLL). In a distributed Lovász Local Lemma instance we are given a set of independent random variables $\mathcal{V}$ and a family $\mathcal{X}$ of (bad) events $\mathcal{E}_1, \ldots, \mathcal{E}_n$ on these variables. Each bad event $\mathcal{E}_i \in \mathcal{X}$ depends on some subset $vbl(\mathcal{E}_i) \subseteq \mathcal{V}$ of the variables. Define the dependency graph $H_{\mathcal{X}} = (\mathcal{X}, \{(\mathcal{E}, \mathcal{E}') \mid vbl(\mathcal{E}) \cap vbl(\mathcal{E}') \neq \emptyset\})$ that connects any two events which share at least one variable. Let Δ_H be the maximum degree in this graph, i.e., each event $\mathcal{E} \in \mathcal{X}$ shares variables with at most $d = \Delta_H$ other events $\mathcal{E}' \in \mathcal{X}$. The Lovász Local Lemma [23] states that $Pr(\bigcap_{\mathcal{E}\in\mathcal{X}} \mathcal{E}) > 0$ if $epd < 1$. In the *distributed LLL problem* each bad event and each variable is assigned to a vertex of the communication network such that all variables that influence a bad event $\mathcal{E}$ that is assigned to a vertex v are either assigned to v or a neighbor of v. The objective is to compute an assignment for the variables such that no bad event occurs. At the end of the computation each vertex v has to know the values of all variables that influence one of its bad events (by definition these variables are assigned to v or one of its neighbors). In all results on distributed LLL in the LOCAL model that are stated below the communication network is identical to the dependency graph H. Most result in the distributed setting require stronger LLL criteria.

Algorithm 1 The algorithm from [20] iteratively resamples all variables of local ID minima of violated events. IDs can be assigned in an adversarial manner. The runtime is with high probability $O(\log_{epd^2} n)$ rounds under LLL criterion $epd^2 < 1$.

```
Initialize a random assignment to the variables
Let $\mathcal{F}$ be the set of bad events under the current variable assignment
while $\mathcal{F} \neq \emptyset$ do
    Let $I = \{A \in \mathcal{F} \mid ID(A) = \min\{ID(B) | B \in N^+_{\mathcal{F}}(A)\}\}$
    Resample $vbl(I) = \cup_{A\in I} vbl(A)$
end
```

Despite its simplicity – the algorithm simply iteratively resamples local ID minima of *violated events*, i.e., bad events that hold under the current variable assignment – Algorithm 1 results in the following theorem.

▶ **Theorem 9** ([20], Algorithm 1). *Given an LLL instance with condition $epd^2 < 1$, the* LOCAL *Algorithm 1 run for $O(\log_{epd^2} n)$ rounds and has error probability $< 1/n$ on any LLL instance with at most n bad events.*

▶ **Definition 10** (Range bounded Lovász Local Lemma). *An instance $(\mathcal{V}, \mathcal{X})$ of the Lovász Local Lemma with bad events $\mathcal{E}_1, \ldots, \mathcal{E}_n$ and dependency graph H is* range bounded *if $|vbl(\mathcal{E}_i)| = \operatorname{poly}\Delta_H$ for each $1 \leq i \leq n$ and the value of each random variable $x \in \mathcal{V}$ can be expressed by at most $O(\log \Delta_H)$ bits.*

Note that most applications of the Lovász Local Lemma in the distributed form are range bounded instances. Algorithm 1 can be executed with the same asymptotic runtime for range bounded LLL instances and uses few random bits while doing so.

▶ **Lemma 11** (Proof deferred to the full version). *For a range bounded LLL instance with condition $epd^2 < 1$ with constant dependency degree there is a randomized* CONGEST *(constant bandwidth) algorithm that runs in $O(\log_{epd^2} n)$ rounds and has error probability $< 1/n$ on any (dependency) graph with at most n nodes. Furthermore, in the algorithm each node only requires access to $O(\log n)$ random values from a bounded range. The unique IDs can be replaced by an acyclic orientation of the edges of the graph.*

The currently fastest randomized algorithm for LLL on bounded degree graphs with polynomial criterion in the LOCAL model is based on two main ingredients. The first ingredient is a deterministic poly $\log n$-round LLL algorithm obtained via a derandomization (cf. [32, 27]) of the $O(\log^2 n)$-round randomized algorithm [39] using the breakthrough efficient network decomposition algorithm [43]. The second ingredient is the (randomized) shattering framework for LLL instances by [24], which *shatters* the graph into small unsolved components of logarithmic size. Then, these components are solved independently and in parallel with the deterministic algorithm.

We follow the same general approach of shattering the graph via the methods of [24], but need to be more careful to obtain a bandwidth efficient algorithm when dealing with the small components. We begin with a lemma that describes how fast we can gather information in a cluster leader with limited bandwidth. Its proof uses standard pipelining techniques (see e.g. [42, Chapter 3] and fills each b-bit messages with as much information as possible.

▶ **Lemma 12** (Token learning). *Let G be a communication graph on n vertices in which each node can send b bits per round on each edge. Assume a cluster collection with Steiner radius β and congestion κ in which each cluster $\mathcal{C}$ is of size at most N and each vertex holds at most x bits of information. Then, in parallel each cluster leader $\ell_{\mathcal{C}}$ can learn the information of each vertex of $\mathcal{C}$ in $O\big(\kappa \cdot (\beta + N \cdot x/b)\big)$ rounds. In the same runtime the leader $\ell_{\mathcal{C}}$ can disseminate x bits of distinct information to each vertex of $\mathcal{C}$.*

Next, we prove our core derandomization result that we use to obtain efficient deterministic algorithms for the small components that arise in the post-shattering phase. It might be of independent interest.

▶ **Lemma 13** (Derandomization). *Consider an LCL problem P, possibly with promises on the inputs. Assume a $T(n)$-round randomized* CONGEST *(bandwidth $b = \Theta(\log n)$) algorithm $\mathcal{A}$ for P with error probability $< 1/n$ on any graph with at most n nodes that uses at most $O(\log n)$ random bits per node.*

Then for any graph on at most N nodes there is a deterministic $T(N) \cdot \operatorname{poly}\log N$ round CONGEST *(with bandwidth $b = \Theta(N)$) algorithm $\mathcal{B}$ to solve P under the same promises, even if the ID space is exponential in N.*

Proof. For an execution of algorithm $\mathcal{A}$ on a graph with N nodes define for each $v \in V$ an indicator variable X_v that equals to 1 if the verification of the solution fails at node v and 0 otherwise. The value of X_v depends on the randomness of nodes in the $(T(N) + r)$ hop neighborhood of v, where $r = O(1)$ is defined by the LCL problem.

We design an efficient algorithm $\mathcal{B}$ that deterministically fixes the (random bits) of each node $v \in V$ such that $X_v = 0$ for all nodes. Thus, executing $\mathcal{A}$ with these random bits solves problem P. During the execution of $\mathcal{B}$ we fix the random bits of more and more nodes. At some point during the execution of algorithm $\mathcal{B}$ let $X \subseteq V$ denote the nodes whose randomness is already fixed and let ϕ_X be their randomness (we formally define ϕ_X later). The crucial invariant that we maintain for each cluster C at all times is the following

$$\textbf{Invariant:} \quad E\left[\sum_{v\in V} X_v \mid \phi_X\right] < 1. \tag{1}$$

The invariant will be made formal in the rest of the proof. Initially (when $X = \emptyset$), it holds with the linearity of expectation and as the error probability of the algorithm implies that $E[X_v] = P(X_v = 1) < 1/n$ holds.

Algorithm $\mathcal{B}$

Compute a weak network decomposition with cluster distance strictly more than $4(T(N)+r)$ with $O(\log N)$ color classes, $\beta = T(N) \cdot \operatorname{poly}\log N$ Steiner tree cluster radius and congestion $\kappa = O(\log N)$. Iterate through the color classes of the decomposition and consider each cluster separately. We describe the process for one cluster $\mathcal{C}$ when processing the i-th color class of the network decomposition. We define three layers according to the distance to vertices in $\mathcal{C}$. Denote the vertices in $\mathcal{C}$ as $W^0_{\mathcal{C}}$, the vertices in $V \setminus W^0_{\mathcal{C}}$ in distance at most $r+T(N)$ from $\mathcal{C}$ as $W^1_{\mathcal{C}}$ and the vertices in $V \setminus (W^0_{\mathcal{C}} \cup W^1_{\mathcal{C}})$ with distance at most $2(r+T(N))$ from $\mathcal{C}$ as $W^2_{\mathcal{C}}$. Define $W_{\mathcal{C}} = W^0_{\mathcal{C}} \cup W^1_{\mathcal{C}} \cup W^2_{\mathcal{C}}$. Note that we have $W_{\mathcal{C}} \cap W_{\mathcal{C}'} = \emptyset$ for two distinct clusters $\mathcal{C}, \mathcal{C}'$ in the same color class of the decomposition due to the cluster distance. Further, even if Lemma 13 is applied to a subgraph H (with size at most N) of a communication network G we have $|W_{\mathcal{C}}| \leq N$, as $W_{\mathcal{C}}$ only contains nodes of H.

Dealing with one cluster $\mathcal{C}$: Extend the Steiner tree of $\mathcal{C}$ to $W_{\mathcal{C}}$ using a BFS, ties broken arbitrarily. Notice that no two BFS trees interfere with each other since we handle different color classes separately and two clusters of the same color are in large enough distance. Use the Steiner tree to assign new IDs (unique only within the nodes in $W_{\mathcal{C}}$)) from range $[N]$ to all nodes in $W_{\mathcal{C}}$. Different clusters use the same set of IDs and each ID can be represented with $O(\log N)$ bits. Each node of $W_{\mathcal{C}}$ learns about the new ID of each of its (at most) Δ neighbors in $W_{\mathcal{C}}$ in one round and uses $O(\Delta \cdot \log N) = O(\log N)$ bits to store its adjacent edges. Using Lemma 12, the cluster leader $\ell_{\mathcal{C}}$ learns the whole topology, inputs and already determined random bits of $G[W_{\mathcal{C}}]$ in $O(\kappa \cdot (\beta + (N \cdot \operatorname{poly}\log N)/b))$ rounds. In this step, the input also includes an acyclic orientation of the edges between vertices of $W_{\mathcal{C}}$, where an edge is oriented from $u \in W_{\mathcal{C}}$ to $v \in W_{\mathcal{C}}$ if the original ID of v is larger than the original ID of v.

For a node v let R_v describe its randomness. When we process cluster $\mathcal{C}$ we determine values $\{r_v \mid v \in \mathcal{C}\}$ for the random variables $\{R_v \mid v \in \mathcal{C}\}$. Let X be a set of vertices v for which we have already determined values r_v. Then we denote $\phi_X = \bigwedge_{v\in X}(R_v = r_v)$. The proof of the following claim is deferred to Appendix A.

▷ Claim 14. Assume that Invariant (1) holds for $X = \bigcup_{\mathcal{C} \text{ has color } < i} \mathcal{C}$ before processing the clusters of color i. Then the cluster leaders $\ell_{\mathcal{C}}$ of all clusters $\mathcal{C}$ of color i, can in parallel, find values $\{r_v \mid v \in \mathcal{C}\}$ such that Invariant (1) holds afterwards for $X' = \bigcup_{\mathcal{C} \text{ has color } \leq i} \mathcal{C}$.

Once the cluster leader $\ell_{\mathcal{C}}$ has fixed the randomness for the vertices in $\mathcal{C}$, it disseminates them to all vertices of its cluster via Lemma 12 and we continue with the next color class of the network decomposition. At the end of the algorithm each node knows the values of its random bits and one can execute $T(N)$ with it, as the initial algorithm $\mathcal{A}$ also works with bandwidth b. Due to the Invariant (1) we obtain that the distributed random bits are such that $E[\sum_{v\in V} X_v \mid \phi_V] < 1$ at the end of the algorithm, but as all X_v's are in $\{0,1\}$ and there is no randomness involved (each vertex is processed in at least one cluster and thus we fix $R_v = r_v$ for each $v \in V$) we obtain that $X_v = 0$ for all $v \in V$, that is, the algorithm does not fail at any vertex.

Runtime: Computing the network decomposition takes $T(N) \cdot \operatorname{poly}\log N$ rounds via Theorem 6. The runtime for processing one color class of the network decomposition is bounded as follows. Collecting the topology and all other information takes $T(N) \cdot \operatorname{poly}\log N$ rounds due to Lemma 12, fixing randomness locally does not require communication. Disseminating the random bits to the vertices of each cluster takes $\operatorname{poly}\log N$ rounds, again due to Lemma 12. As there are only $\log N$ color classes the runtime for computing good random bits for all nodes is $T(N) \cdot \operatorname{poly}\log N$. Executing algorithm $\mathcal{A}$ with these random bits takes $T(N)$ rounds. ◀

▶ **Lemma 15.** *There is a deterministic LLL algorithm for range bounded LLL instances with constant dependency degree d and LLL criterion $epd^2 < 1$ that runs in $O(\operatorname{poly}\log\log n)$ rounds on any graph with at most n nodes if the communication bandwidth is $b = \Theta(n)$.*

Proof. Plug the randomized algorithm of Lemma 11 into the derandomization result of Lemma 13. We can apply the lemma because the feasibility of a computed solution can be checked in 1 round in CONGEST. ◀

Proof of Theorem 1. The goal is to apply the shattering framework of [24]. The pre-shattering phase takes $\operatorname{poly}\Delta + O(\log^* n)$ rounds and afterwards all events with an unset variable induce small components of size $N = \operatorname{poly}(\Delta, \log n)$. Furthermore, each of the small components also forms an LLL instance, but with a slightly worse LLL criterion. Then we apply Lemma 15 to solve all small components in parallel. Next, we describe these steps in detail. The pre-shattering phase begins with computing a distance-2 coloring of the dependency graph H with $O(\Delta^2)$ colors, i.e., a coloring in which each color appears only once in each inclusive neighborhood. In the LOCAL model this takes $O(\log^* n)$ rounds with Linial's algorithm [37]. In the CONGEST model we can compute such a coloring in $O(\operatorname{poly}\Delta_H \cdot \log^* n) = O(\log^* n)$ rounds.

Next, we iterate through the $O(\Delta^2)$ color classes to set some of the variables. The unset variables either have the status *frozen* or *non-frozen*. At the beginning all variables are unset and non-frozen. We iterate through the $O(\Delta_H^2)$ color classes and process all vertices (events) of the same color class in parallel. Each non-frozen variable of a processed event $\mathcal{E}$ is sampled; then the node checks how the conditional probabilities of neighboring events have changed. As the LLL is range bounded this step can be implemented in $O(1)$ rounds. If there is an event $\mathcal{E}'$ (possibly $= \mathcal{E}$) whose probability has increased to at least $p' = \sqrt{p}$, its variables are unset and all variables of event $\mathcal{E}'$ are frozen. The next three observations are proven in [24] and capture the properties of this pre-shattering phase.

▶ **Observation 16.** *For each event $\mathcal{E} \in \mathcal{X}$, the probability of $\mathcal{E}$ having at least one unset variable is at most $(d+1)\sqrt{p}$. Furthermore, this is independent of events that are further than 2 hops from $\mathcal{E}$.*

The following result follows with Observation 16 and the by now standard shattering lemma. We do not discuss the details as it has been done in [24].

▶ **Observation 17** (Small components)**.** *The connected components in H induced by all events with at least one unset variable are w.h.p. in n of size $N = \operatorname{poly}(\Delta_H) \cdot \log n = O(\log n)$.*

The following observation holds as each event with an unset variable fails at most with probability $p' = \sqrt{p}$ when its variables are frozen. It is proven in [24].

▶ **Observation 18.** *The problem on each connected component induced by events with at least one unset variable is an LLL problem with criterion $p'(d+1) < 1$.*

To complete our proof we apply the deterministic algorithm of Lemma 15 on each component in parallel. Let U be the set of nodes that have its random bits not yet determined. Note that any vertex in U is part of one component. For each $u \in U$ the values of already determined random bits in the r-hop ball around u are included in u's input – many nodes already determine their random bits in the pre-shattering phase. Even conditioned on the random bits determined in the pre-shattering phase (formally this provides a promise to the inputs), each instance is a range bounded LLL on a bounded degree graph with at most N nodes and the standard CONGEST bandwidth is $b = \Omega(\log n) = \Omega(N)$, the runtime is $\operatorname{poly}\log N = \operatorname{poly}\log\log n$. All instances can be dealt with independently since by definition, the connected unsolved components share no events nor variables. ◀

The celebrated result by [19] says that an LCL problem either cannot be solved faster than in $\Omega(\log n)$ rounds or can be solved with an LLL algorithm. In our work, we show that their proof can be easily extended to work with range bounded LLLs.

▶ **Lemma 19** ([19], Proof deferred to the full version)**.** *Any LCL problem that can be solved with a randomized $o(\log n)$-round* LOCAL *algorithm with error probability $< 1/n$ on any graph with at most n nodes can be solved via the following procedure: Create a bounded range LLL instance with LLL criterion $p(ed)^{100} < 1$, solve the instance, and run a constant time deterministic* LOCAL *algorithm that uses the solution of the LLL instance as input.*

Lemma 19 of [19] has been used in several other works, e.g., in [3, 13]. Note that [3] sets up the same LLL with the purpose of designing an efficient CONGEST algorithm for it; however, [3] is restricted to the setting where the input graph is a tree which allows for very different methods of solving the respective LLL and admits even an $O(\log\log n)$-round algorithm. We combine Lemma 19 with our LLL algorithm in Theorem 1 to prove Corollary 2.

▶ **Corollary 2.** *There is no LCL problem with randomized complexity strictly between* $\operatorname{poly}\log\log n$ *and* $\Omega(\log n)$ *in the* CONGEST *model.*

The results in this section imply randomized $\operatorname{poly}\log\log n$-round algorithms for classic problems such as Δ-coloring on constant degree graphs (as Δ-coloring is an LCL which has an $o(\log n)$-round LOCAL algorithm [29] the result follows along the same lines as Corollary 2) and various defective coloring variants for constant degree graphs by modeling them as range bounded LLLs and applying Theorem 1, see [20] for various such problems and how they can be modeled as LLLs.

Our network decomposition algorithm with few colors in combination with the LOCAL model LLL algorithm from [24] provides the following theorem. Due to the unconstrained message size in the LOCAL model, it does not require the LLL instances to be range bounded.

▶ **Theorem 20.** *Let $\lambda \in \mathbb{N}$ be constant. There exists a deterministic $n^{2/\lambda}$* $\operatorname{poly}\log n$ *round* LOCAL *algorithm for LLL instances with criterion $p(ed)^{\lambda} < 1$.*

References

1 Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th Symp. on Found. of Computer Science (FOCS)*, pages 364–369, 1989.

2 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. In *32nd International Symposium on Distributed Computing, DISC*, pages 9:1–9:16, 2018. doi:10.4230/LIPIcs.DISC.2018.9.

3 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In *International Symposium on Distributed Computing DISC*, 2021.

4 Alkida Balliu, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 1307–1318, 2018. doi:10.1145/3188745.3188860.

5 Philipp Bamberger, Fabian Kuhn, and Yannic Maus. Efficient deterministic distributed coloring with small bandwidth. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 243–252, 2020. doi:10.1145/3382734.3404504.

6 Leonid Barenboim. On the locality of some np-complete problems. In *Automata, Languages, and Programming - 39th International Colloquium (ICALP)*, pages 403–415, 2012. doi:10.1007/978-3-642-31585-5_37.

7 Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool Publishers, 2013.

8 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theor. Comput. Sci.*, 751:2–23, 2018. doi:10.1016/j.tcs.2016.07.005.

9 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM*, 63(3):20:1–20:45, 2016.

10 József Beck. An algorithmic approach to the Lovász local lemma. *Random Structures & Algorithms*, 2(4):343–365, 1991.

11 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A Lower Bound for the Distributed Lovász Local Lemma. In *ACM Symposium on Theory of Computing (STOC)*, 2016.

12 Sebastian Brandt, Christoph Grunau, and Václav Rozhon. Generalizing the sharp threshold phenomenon for the distributed complexity of the lovász local lemma. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 329–338, 2020. doi:10.1145/3382734.3405730.

13 Sebastian Brandt, Christoph Grunau, and Václav Rozhon. The randomized local computation complexity of the lovász local lemma. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 307–317. ACM, 2021. doi:10.1145/3465084.3467931.

14 Sebastian Brandt, Yannic Maus, and Jara Uitto. A sharp threshold phenomenon for the distributed complexity of the lovász local lemma. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed (PODC)*, pages 389–398, 2019. doi:10.1145/3293611.3331636.

15 Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. *Distributed Comput.*, 33(3-4):349–366, 2020. doi:10.1007/s00446-020-00376-1.

16 Yi-Jun Chang and Mohsen Ghaffari. Strong-diameter network decomposition. *the Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, abs/2102.09820, 2021. arXiv:2102.09820.

17 Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The complexity of distributed edge coloring with small palettes. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2633–2652, 2018. doi:10.1137/1.9781611975031.168.

18 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 615–624, 2016.

19 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput.*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

20 Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed Algorithms for the Lovász Local Lemma and Graph Coloring. *Distributed Computing*, 30(4):261–280, 2017. doi:10.1007/s00446-016-0287-6.

21 Janosch Deurer, Fabian Kuhn, and Yannic Maus. Deterministic distributed dominating set approximation in the CONGEST model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 94–103, 2019. doi:10.1145/3293611.3331626.

22 Michael Elkin and Ofer Neiman. Distributed strong diameter network decomposition. In *Proc. 35th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 211–216, 2016.

23 Paul Erdös and László Lovász. Problems and Results on 3-chromatic Hypergraphs and some Related Questions. *Colloquia Mathematica Societatis János Bolyai*, pages 609–627, 1974.

24 Manuela Fischer and Mohsen Ghaffari. Sublogarithmic Distributed Algorithms for Lovász Local Lemma, and the Complexity Hierarchy. In *the Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, pages 18:1–18:16, 2017. doi:10.4230/LIPIcs.DISC.2017.18.

25 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proc. 27th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 270–277, 2016.

26 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *the Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2021. arXiv:2007.08253.

27 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On derandomizing local distributed algorithms. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 662–673, 2018. doi:10.1109/FOCS.2018.00069.

28 Mohsen Ghaffari, David G. Harris, and Fabian Kuhn. On Derandomizing Local Distributed Algorithms. In *the Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 662–673, 2018.

29 Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, and Yannic Maus. Improved distributed δ-coloring. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 427–436, 2018. URL: https://dl.acm.org/citation.cfm?id=3212764.

30 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In *32nd International Symposium on Distributed Computing (DISC)*, pages 29:1–29:17, 2018. doi:10.4230/LIPIcs.DISC.2018.29.

31 Mohsen Ghaffari and Fabian Kuhn. On the use of randomness in local distributed graph algorithms. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 290–299, 2019. doi:10.1145/3293611.3331610.

32 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 784–797, 2017. doi:10.1145/3055399.3055471.

33 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 784–797. ACM, 2017.

34 Mohsen Ghaffari and Julian Portmann. Improved network decompositions using small messages with applications on mis, neighborhood covers, and beyond. In *33rd International Symposium on Distributed Computing (DISC)*, pages 18:1–18:16, 2019. doi:10.4230/LIPIcs.DISC.2019.18.

35 Magnús M. Halldórsson, Fabian Kuhn, and Yannic Maus. Distance-2 coloring in the CONGEST model. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 233–242, 2020. doi:10.1145/3382734.3405706.

36 Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Alexandre Nolin. Coloring fast without learning your neighbors' colors. In *34th International Symposium on Distributed Computing (DISC)*, pages 39:1–39:17, 2020. doi:10.4230/LIPIcs.DISC.2020.39.

37 Nati Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

38 Nati Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.

39 Robin A. Moser and Gábor Tardos. A Constructive Proof of the General Lovász Local Lemma. *J. ACM*, pages 11:1–11:15, 2010.

40 M. Naor and L. Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.

41 Alessandro Panconesi and Aravind Srinivasan. On the complexity of distributed network decomposition. *Journal of Algorithms*, 20(2):581–592, 1995.

42 David Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

43 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *ACM Symposium on Theory of Computing (STOC)*, pages 350–363, 2020.

A Proof of Claim 14

▷ Claim 14. Assume that Invariant (1) holds for $X = \bigcup_{\mathcal{C} \text{ has color } < i} \mathcal{C}$ before processing the clusters of color i. Then the cluster leaders $\ell_{\mathcal{C}}$ of all clusters $\mathcal{C}$ of color i, can in parallel, find values $\{r_v \mid v \in \mathcal{C}\}$ such that Invariant (1) holds afterwards for $X' = \bigcup_{\mathcal{C} \text{ has color } \leq i} \mathcal{C}$.

If we considered just a single cluster $\mathcal{C}$, independently from all other clusters with color i, then there are choices for the values of r_v satisfying the claim due to the law of total probability, e.g., as used in the method of conditional expectation, and as $E\big[\sum_{v \in V} X_v \mid \phi_X\big] < 1$ holds before we fix any randomness of vertices inside $\mathcal{C}$.

Proof. The only information that cluster leaders need to perform the necessary calculations is information on the topology of $G[W_{\mathcal{C}}]$, inputs to nodes in $W_{\mathcal{C}}$, the relative order of adjacent IDs of nodes in $G[W_{\mathcal{C}}]$ and already determined randomness of nodes in $X \cap W_{\mathcal{C}}$.

We show that the randomness of all vertices in clusters with color i can be fixed with this knowledge. To this end the cluster leader $r_{\mathcal{C}}$ fixes the randomness $\phi_{\mathcal{C}}$ such that

$$E\big[\sum_{v \in W_{\mathcal{C}}} X_v \mid \phi_{\mathcal{C}} \wedge \phi_{X \cap W_{\mathcal{C}}}\big] \leq E\big[\sum_{v \in W_{\mathcal{C}}} X_v \mid \phi_{X \cap W_{\mathcal{C}}}\big]$$

holds. Such a choice for $\phi_{\mathcal{C}}$ exists, again due to the law of total probability, and $\ell_{\mathcal{C}}$ has full information to compute such values as all values in the inequality only depend on information that vertices in $W_{\mathcal{C}}$ sent to $\ell_{\mathcal{C}}$. In particular, the randomness of nodes in $\mathcal{C}$ only influences the random variables X_v of nodes in $W^0_{\mathcal{C}} \cup W^1_{\mathcal{C}}$ and computing this influence only requires knowledge from $W^0_{\mathcal{C}} \cup W^1_{\mathcal{C}} \cup W^2_{\mathcal{C}}$.

After all cluster leaders of clusters with color i fix the randomness of the vertices in their clusters according to the above method the invariant is still satisfied since the randomness R_v for $v \in \mathcal{C}$ does not influence the random variable X_u for $u \in V \setminus W_{\mathcal{C}}$. More formally, let X be the set of vertices with fixed randomness before we process the i-th color class of clusters and let Y be the set of vertices whose randomness we fix when processing the i-th color class. Let $X' = X \cup Y$. Let $W = \bigcup_{\mathcal{C} \text{ has color } i} W_{\mathcal{C}}$. Recall that $W_{\mathcal{C}} \cap W_{\mathcal{C}'} = \emptyset$ for $\mathcal{C} \neq \mathcal{C}'$. Due to the linearity of expectation and the aforementioned reasons we obtain

$$\begin{aligned}
E\left[\sum_{v\in V} X_v \mid \phi_{X'}\right] &= E\left[\sum_{v\in V\setminus W} X_v \mid \phi_{X'}\right] + \sum_{\mathcal{C}\text{ has color } i} E\left[\sum_{v\in W_{\mathcal{C}}} X_v \mid \phi_{X'\cap W_{\mathcal{C}}}\right] \\
&\leq E\left[\sum_{v\in V\setminus W} X_v \mid \phi_{X}\right] + \sum_{\mathcal{C}\text{ has color } i} E\left[\sum_{v\in W_{\mathcal{C}}} X_v \mid \phi_{X\cap W_{\mathcal{C}}}\right] \\
&= E\left[\sum_{v\in V} X_v \mid \phi_{X}\right] < 1.
\end{aligned}$$

All clusters can be processed in parallel as their distance is strictly more than $4(T(N)+r)$ and the choices in cluster $\mathcal{C}$ do not change the probability for $X_v = 1$ for $v \in \mathcal{C}' \neq \mathcal{C}$. $\triangleleft$

Optimal Communication Complexity of Authenticated Byzantine Agreement

Atsuki Momose ✉
Nagoya University, Aichi, Japan
Intelligent Systems Laboratory, SECOM CO.,LTD., Tokyo, Japan

Ling Ren ✉
University of Illinois at Urbana-Champaign, Urbana, IL, USA

Abstract

Byzantine Agreement (BA) is one of the most fundamental problems in distributed computing, and its communication complexity is an important efficiency metric. It is well known that quadratic communication is necessary for BA in the worst case due to a lower bound by Dolev and Reischuk. This lower bound has been shown to be tight for the unauthenticated setting with $f < n/3$ by Berman et al. but a considerable gap remains for the authenticated setting with $n/3 \leq f < n/2$.

This paper provides two results towards closing this gap. Both protocols have a quadratic communication complexity and have different trade-offs in resilience and assumptions. The first protocol achieves the optimal resilience of $f < n/2$ but requires a trusted setup for threshold signature. The second protocol achieves near optimal resilience $f \leq (1/2 - \varepsilon)n$ in the standard PKI model.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Byzantine Agreement, Communication Complexity, Lower Bound

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.32

Acknowledgements We would like to thank Zhuolun Xiang for helpful feedback.

1 Introduction

Byzantine Agreement (BA) is one of the most fundamental problems in distributed algorithms [21]. It also serves as an important building block in cryptography and distributed systems. At a high level, Byzantine agreement is the problem for n parties to agree on a value, despite that up to f of them may behave arbitrarily (called Byzantine faults). Arguably the most important efficiency metric of Byzantine Agreement is the communication complexity, since communication will be the bottleneck in applications like state machine replication and cryptocurrency when there is a large number of parties.

Dolev and Reischuk proved that a quadratic number of messages are necessary for any perfectly secure BA protocol. More formally, they showed that even in the authenticated setting (i.e., assuming public key infrastructure and ideal digital signature), any BA protocol with perfect security (i.e., all executions are correct) has at least one execution where quadratic number of messages are sent by honest parties. The tightness of this lower bound was partially established by Berman et al. in the unauthenticated setting with $f < n/3$. However, for decades, the best known protocol for the authenticated setting (with $f \geq n/3$) remains the classic Dolev-Strong protocol [14] [1], which uses quadratic messages but cubic communication. The reason is that in Dolev-Strong, the messages can contain up to $f + 1$ signatures. Therefore, the optimal worst-case communication complexity of authenticated BA with $f \geq n/3$ has remained an open problem for decades.

[1] Dolev-Strong solves a related problem called Byzantine broadcast, but it is easy to transform it into a BA protocol.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 32; pp. 32:1–32:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Upper bounds for worst-case communication complexity of Byzantine agreement with assumptions preserving the quadratic lower bound. ε is any positive constant.

protocol	model	communication	resilience
Berman et al. [6]	unauthenticated	$O(n^2)$	$f < n/3$
Dolev-Strong [14]	authenticated	$O(\kappa n^2 + n^3)$ a)	$f < n/2$ b)
this paper	threshold signature	$O(\kappa n^2)$	$f < n/2$
this paper	authenticated	$O(\kappa n^2)$	$f \leq (\frac{1}{2} - \varepsilon)n$

a) The original Dolev-Strong protocol solves BB but can be easily converted into a BA protocol with an initial round to multicast the inputs. Using a multi-signature with a list of signer identities attached, the protocol achieves $O(\kappa n^2 + n^3)$.

b) Although the original Dolev-Strong BB protocol tolerates $f < n$ faults, converting it to a BA protocol decreases the fault tolerance to $f < n/2$, which is optimal for authenticated BA.

This paper provides two results that help close this gap. More specifically, we show the following two theorems. Note that when $f \geq n/3$, it is necessary to adopt the synchronous and authenticated setting. Under asynchrony [18], partial synchrony [15], or the unauthenticated setting [17], BA is impossible for $f \geq n/3$.

▶ **Theorem 1.** *Assuming a threshold signature scheme, there exists a Byzantine agreement protocol with $O(\kappa n^2)$ communication complexity tolerating $f < n/2$ faults where n is the number of parties and κ is a security parameter.*

▶ **Theorem 2.** *Assuming a digital signature scheme with a public-key infrastructure, there exists a Byzantine agreement protocol with $O(\kappa n^2)$ communication complexity tolerating $f \leq (\frac{1}{2} - \varepsilon)n$ faults where n is the number of parties, κ is a security parameter, and ε is any positive constant.*

As we can see, the above two results achieve quadratic worst-case communication with different trade-offs. The first result achieves the optimal resilience $f < n/2$ but relies on a trusted setup due to the use of threshold signature. On the other hand, the second result is in the standard PKI model, but there is a small gap in the resilience.

Tightness with respect to the Dolev-Reischuk lower bound. In the Byzantine agreement and Byzantine fault tolerance literature, it is common and convenient to abstract signatures as ideal oracles and focus on the aspect of distributed computing [21, 14, 13, 9]. The rational is that modern cryptography has given us solid understandings and confidence about digital signatures, and that the probability that an adversary breaks a signature scheme is too small to be a concern.

When we abstract digital signatures and threshold signatures as ideal oracles with perfect security, our two results match the quadratic worst-case communication lower bound established by Dolev and Reischuk. Table 1 compares our results to the current landscape of worst-case communication complexity of perfectly secure BA.

On this note, it is very important to note that the Dolev-Reischuk lower bound applies to any protocol that is perfectly secure, even if the protocol has access to ideal digital signature and threshold signature oracles. With ideal digital signature and threshold signature oracles, our protocols are perfectly secure. On the other hand, there exist in the literature sub-quadratic BA protocols [20, 10, 1, 11] that use randomization techniques and allow a negligible fraction of the executions to fail. These protocols do not provide perfect security even if we assume their randomization primitives are ideal. Naturally, they do not address the tightness of the Dolev-Reischuk lower bound.

We also remark that while it is possible to circumvent the Dolev-Reischuk lower bound by allowing a small failure probability, it should be clear that if the only source of failure in a protocol comes from imperfect cryptographic primitive, the protocol will not be able to circumvent the lower bound, because upgrading the cryptographic primitive from imperfect security to perfect security (at no extra costs) only strengthens the protocol.

Comparing with state-of-the-art BA solutions. Although our primary motivation of this study is to show the tightness of the quadratic lower bound of Dolev-Reischuk, the second result in Theorem 2 has some advantage even over state-of-the-art BA protocols with assumptions that are not subject to the Dolev-Reischuk bound. To the best of our knowledge, our second protocol is the first to achieve the following three properties simultaneously under the standard PKI model: (1) near-optimal resilience of $f \leq (\frac{1}{2} - \varepsilon)n$, (2) security against an adaptive adversary, (3) expected sub-cubic communication complexity. In fact, our protocol achieves worst-case quadratic communication and is secure against a strongly rushing (defined in [1]) adaptive adversary. The works of Berman et al. [6] and King-Saia [20] achieve (sub-)quadratic communication and adaptive security but tolerate only $f < n/3$. Abraham et al. [2, 1] achieve (sub-)quadratic communication and adaptive security under $f \leq (\frac{1}{2} - \varepsilon)n$, but require some trusted setup assumption due to the use of threshold signature or verifiale random functions. Tsimos et al. [28] recently achieve nearly-quadratic communication in the standard PKI model for $f \leq (1 - \varepsilon)n$ (for broadcast), but it is secure only against a static adversary.

Organization. The rest of the paper is organized as follows. In the rest of this section, we briefly review related work and give an overview of the techniques we use to achieve our two results. Section 2 introduces definitions, models and notations. Section 3 introduces the recursive framework to get a BA protocol with quadratic communication including the definition of GBA primitive. Section 4 presents two GBA protocols to instantiate two BA protocols with different trade-offs to complete our results. Finally, we discuss future directions and conclude the paper in Section 5.

1.1 Technical Overview

Abstracting the recursive framework of Berman et al. To obtain the results, we revisit the Berman et al. [6] protocol. At a high level, Berman et al. is a recursive protocol: it partitions parties into two halves recursively until they reach a small instance with sufficiently few (e.g., a constant number of) participants. Since the upper bound on the fraction of faults 1/3 is preserved in at least one of two halves, the "correct" half directs the entire parties to reach an agreement. If the communication except the two recursive calls is quadratic, the communication complexity of the entire protocol is also quadratic. The challenge is to prevent an "incorrect" run of recursive call (in a half with more than 1/3 faults) from ruining the result. Berman et al. solve this problem with a few additional rounds of communication called "universal exchange" before each recursive call. It helps honest parties stick to a value when all honest parties already agree on the value, thus preventing an incorrect recursive call from changing the agreed-upon value.

Back to our setting of $f \geq n/3$, we will use the recursive framework of Berman et al.. However, the universal exchange step of Berman et al. relies on a quorum-intersection argument, which only works under $f < n/3$. To elaborate, the quorum size can be at most $n-f$; two quorums of size $n-f$ intersect at $2(n-f)-n = n-2f$ parties; for this intersection to contain at least one honest party, it requires $n-2f > f$, or equivalently $f < n/3$.

To achieve our goal of BA with $f \geq n/3$, we observe that the functionality achieved by the universal exchange can be abstracted as a primitive called graded Byzantine agreement (GBA), which we formally define in Section 3. If we can construct a GBA with quadratic communication and plug it into the recursive framework, we will obtain a BA protocol with quadratic communication. Thus, it remains to construct quadratic GBA.

Two constructions of Graded BA with different trade-offs. As the name suggests, the GBA primitive shares some similarities with graded broadcast studied in [16, 19, 2], but it is harder to construct due to the fact that every party has an input. This can be addressed in two ways, leading to our two constructions.

The first method way is to resort to the (well-established) use of threshold signatures [7, 29]. Roughly, a threshold signature condenses a quorum of $n - f = \Omega(n)$ votes into a succinct proof of the voting result. This way, a verifiable voting result can be multicasted to all parties using quadratic total communication (linear per node). This achieves Theorem 1 and requires a trusted setup for threshold signature.

Next, we try to construct a quadratic GBA without trusted setup or threshold signature scheme. This turns out to be much more challenging. Naïvely multicasting the voting result would require quadratic communication per node (cubic in total) since the voting result consists of a linear number of votes. To get around this problem, we replace the multicast step with communication through an expander graph with constant degree. As each party transmits the voting result to only a constant number of neighbors, the communication is kept quadratic in total even though the voting result consists of a linear number of votes. Our key observation is that even though some of the honest parties may fail to receive or transmit the voting result (because all their neighbors are corrupted), as long as a small but linear number of honest parties transmit the voting result, the good connectivity of the expander helps prevent inconsistent decisions between honest parties. In order to verify a linear number of honest parties actually transmit, a quorum of $n - f$ parties who claim to have transmitted should contain at least a linear number of honest parties, which results in the gap of ϵn in the resilience in Theorem 2.

1.2 Related Work

Byzantine Agreement was first introduced by Lamport et al. [26, 21]. Without cryptography (i.e., the unauthenticated setting), BA can be solved if and only if $f < n/3$. Assuming a digital signature scheme with a public-key infrastructure (i.e., the authenticated setting), BA can be solved if and only if $f < n/2$. Lamport et al. gave BA protocols for both settings, but they both require exponential communication. Later, polynomial communication protocols were shown in both settings. In particular, Dolev and Strong [14] showed a $O(\kappa n^3)$ communication protocol for the authenticated setting and Dolev et al. [12] showed a $O(n^3 \log n)$ communication protocol for the unauthenticated setting. For the unauthenticated setting, Berman et al. further reduced the communication to $O(n^2)$, matching a lower bound established by Dolev and Reischuk [13], which states that any deterministic protocol with perfect security even in the authenticated setting must incur $\Omega(n^2)$ communication complexity. A recent work called HotStuff [29] can be modified [27] to achieve $O(\kappa n^2)$ communication with $f < n/3$ for the authenticated setting.

We also mention several orthogonal lines of work. Some works known as extension protocols [8, 24, 25, 22] achieve an optimal $O(nl)$ communication complexity for sufficiently long inputs of size l using the BA oracle for short inputs. When the input size is small, e.g., $l = O(1)$, the communication complexity degenerates to that of the underlying BA oracle. Our work provides improved oracles for these protocols.

Another line of works study protocols with sub-quadratic communication [20, 10, 1]. The idea is to select a random and unpredictable subset of parties to run the protocol (often using cryptographic primitives such as verifiable random function). Even after assuming ideal common randomness, these protocols will still have a small fraction of insecure executions, and are thus not subject to the Dolev-Reischuk lower bound. In contrast, we only use (threshold) signatures for message authentication. Once we assume ideal (threshold) signatures, our protocols are perfectly secure and are subject to the Dolev-Reischuk.

Other works study protocols with expected quadratic communication protocols [16, 19, 7, 23, 4, 2]. These protocols can require super-quadratic communication in the worst-case.

2 Preliminaries

Execution model. We define a protocol as an algorithm for a set of parties. There are a set of n parties, of which at most $f < n$ are Byzantine faulty and behave arbitrarily. We assume $f = \Theta(n)$. All presented protocols are secure against f adaptive corruption that can happen anytime during the protocol execution. Moreover, we assume a strongly rushing adaptive adversary [2, 1] who can corrupt parties in a round after seeing the messages they sent in that round and immediately delete those messages from network before they reach other parties. A party that is not faulty throughout the execution is said to be honest and faithfully execute the protocol. We use the term *quorum* to mean the minimum number of all honest parties, i.e., $n - f$. A protocol proceeds in synchronous rounds. If an honest party sends a message at the beginning of some round, an honest recipient receives the message at the end of that round.

Ideal (threshold) signatures. As mentioned, our two results, after assuming an ideal signatures and threshold signatures, address the tightness of the Dolev-Resichuk lower bound. We define the interface of signature and threshold signature oracles.

▶ **Definition 3** (Digital signature). *A digital signature oracle provides the following interfaces:*

- $\sigma \leftarrow \mathsf{Sign}_r(x)$. *Party r can invoke this interface to obtain a signature σ by party r on message x.*
- $b \leftarrow \mathsf{Verify}(\sigma, x, r)$. *Any party can invoke this interface to check whether σ is a signature by party r on message x.*

The oracle satisfies the following property.

- *For any σ, x, r, $\mathsf{Verify}(\sigma, x, r)$ outputs $b = 1$ if and only if $\mathsf{Sign}_r(x)$ has been queried by party r and the output is σ.*

The above property ensures *correctness*, i.e., correctly generated signatures are always verified, and *unforgeability*, i.e., no one other than party r can generate a signature for party r. For simplicity, we use $\langle x \rangle_r$ to denote a signed message x by party r, i.e., $\langle x \rangle_r = (x, \sigma)$ where $\sigma = \mathsf{Sign}_r(x)$ Any party can verify a signed message $\langle x \rangle_r = (x, \sigma)$ by querying $\mathsf{Verify}(\sigma, x, r)$.

▶ **Definition 4** ((t,n)-threshold signature)**.** *Each party r have access to the (t,n)-threshold signature oracle that provides the following interfaces, where $t < n/2$ is a given threshold.*

- $\sigma \leftarrow \mathsf{Sign}_r(x)$. *Party r can invoke this interface to obtain a signature share σ by party r on message x.*
- $b \leftarrow \mathsf{VerifyShare}(\sigma, x, r)$. *Any party can invoke this interface to check whether σ is a signature share by party r on message x.*
- $\Sigma \leftarrow \mathsf{Combine}(x, \{\sigma_1, ..\sigma_t\}, \{r_1, ..r_t\})$. *Any party can invoke this interface to combine a set of signature shares $\{\sigma_1, ..\sigma_t\}$ on the message x from t different parties $\{r_1, ..r_t\}$ into a threshold signature Σ.*
- $b \leftarrow \mathsf{Verify}(x, \Sigma)$. *Any party can invoke this interface to check whether Σ is a threshold signature generated from valid t signature shares.*

The oracle satisfies the following properties.

- *For any σ, x, r, $\mathsf{VerifyShare}(\sigma, x, r)$ outputs $b = 1$ if and only if $\mathsf{Sign}_r(x)$ has been queried by party r and the output is σ.*
- *For any x, $\mathsf{Verify}(x, \Sigma)$ outputs $b = 1$ if and only if there exist $\{\sigma_1, ..\sigma_t\}$ and $\{r_1, ..r_t\}$ such that for all $1 \leq i \leq t$, $\mathsf{VerifyShare}(\sigma_i, x, r_i) = 1$, and $\mathsf{Combine}(x, \{\sigma_1, ..\sigma_t\}, \{r_1, ..r_t\})$ has been queried by a party and the output is Σ.*

These two properties together satisfy the correctness and unforgeability properties of the signature shares and threshold signatures as before, and in addition a *robustness* property, i.e., t valid signature shares can always be combined into a valid threshold signature.

For simplicity, we use the same notation $\langle x \rangle_r$ as in digital signature to denote a tuple of message x and a signature share $\sigma \leftarrow \mathsf{Sign}_r(x)$. Each party r verifies a signature share $\langle x \rangle_r = (x, \sigma)$ by querying $\mathsf{VerifyShare}(\sigma, x, r)$. A set of $\langle x \rangle_*$ from t different parties can be combined into a threshold-signed x and verified by any party, using the $\mathsf{Combine}$ and Verify interfaces.

Setup assumptions. In practice, the above oracles are realized with negligible error with a PKI setup or trusted setup. The currently known threshold signature schemes require a trusted dealer who generates all public and private keys for all parties and a group public key to verify a combined full signature, henceforth we call it trusted setup. The digital signature requires the standard PKI setup and does not require any trusted setup beyond that. In that case, each party independently generates a pair of public and private keys without any extra assumption.

The Dolev-Reischuk lower bound. The Dolev-Reischuk lower bound holds (without any modification) even with ideal (threshold) signature oracles. We also note that the Dolev-Reischuk lower bound, which was originally proved for deterministic protocols, can be extended to randomized protocols as well. More precisely, any BA protocol (either deterministic or randomized) cannot simultaneously enjoy perfect security and sub-quadratic worst-case communication complexity. This has been observed and briefly mentioned in [20] and we show a proof for completeness.

▶ **Theorem 5.** *There does not exist a (either deterministic or randomized) BA protocol with worst-case communication complexity of at most $f^2/4$ that is perfectly secure.*

Proof. Suppose for the sake of contradiction that there exists such a protocol P. If P is randomized, we can transform P into a deterministic protocol P^* by fixing the output of the all random coin tossing to 0. Since P if perfectly secure and has at most $f^2/4$ communication cost in the worst case, P^* is a deterministic BA protocol that is perfectly secure and has at most $f^2/4$ communication complexity. This contradicts the original Dolev-Reischuk lower bound [13]. ◀

Give this more general lower bound, our protocols, regardless of whether or not ideal digital signatures and threshold signatures are considered deterministic or randomized, are subject to the quadratic worst-case lower bound.

A remark on complexity metrics. The communication complexity of a protocol is the maximum number of bits sent by all honest parties combined across all executions. Since all messages in our protocols are signed, we use the signature size κ as the unit of measure for communication. We assume the size of any input value is on the order of κ. The Dolev-Reichuk lower bound, however, is in terms of the number of messages. With no assumption on the message size, this leaves a gap of κ in the upper and lower bounds. If we further assume that every message in authenticated protocols is signed, then the bounds match. It is an interesting open problem whether we can design an authenticated protocol that leaves most of the messages *unsigned* to do better than $O(\kappa n^2)$.

Byzantine Agreement. In Byzantine Agreement (BA), each party has an input value, and all parties try to decide on the same value. The requirement of BA is defined as follows.

▶ **Definition 6** (Byzantine Agreement (BA)). *A Byzantine agreement protocol must satisfy the following properties.*

1. *consistency: if two honest parties r and r' decide values v and v', then $v = v'$.*
2. *termination: every honest party decides a value and terminates.*
3. *validity: if all honest parties have the same input value, then all honest parties decide that value.*

Although our main focus of this paper is BA, we also mention a closely related problem called Byzantine broadcast (BB). In BB, a designated sender has an input to broadcast to all parties, and all parties try to decide on the same value. The requirement of BB is defined as follows.

▶ **Definition 7** (Byzantine Broadcast (BB)). *A Byzantine broadcast protocol must satisfy the following properties.*

1. *consistency: same as above.*
2. *termination: same as above.*
3. *validity: if the sender is honest, then all honest parties decide the sender's value.*

It is easy to transform a BA protocol into a BB protocol preserving the same resilience and quadratic communication complexity by having an initial round for the sender to broadcast its input value before starting the BA protocol [21]. As the Dolev-Reischuk lower bound holds for both BA and BB, our results establish the tightness of the quadratic communication complexity for BB as well (though the resilience $f < n/2$ is not optimal for BB, which is possible under any $f < n$).

3 Recursive Framework of Byzantine Agreement with Quadratic Communication

This section reviews the recursive framework to construct a BA protocol with quadratic communication introduced by Berman et al. [6] for $f < n/3$, and making it works for $f < n/2$.

Dissecting Berman et al. In the Berman et al. protocol, parties are partitioned into two halves, and each half runs the BA protocol recursively in sequential order. The partition continues until we reach a BA instance with a constant number of parties, where using any inefficient BA protocol will not impact the overall complexity. At each recursive step, additional quadratic communication is incurred besides the two recursive BA calls. It is not hard to see that the overall communication complexity is quadratic.

Since the fraction of faults in the entire parties is less than 1/3, one of two halves also has faults of less than 1/3 and thus achieve a "correct" BA. However, even if the first committee is correct, the potential incorrect second BA instance may "ruin" the result of the first one. To prevent this, parties run a few rounds of preprocessing steps called "universal exchange" in Berman et al. before each recursive BA call. The universal exchange step helps parties "stick to" a value (ignoring the recursive BA output) if all honest parties already agree on that value. In more detail, if the first run of recursive BA is correct and all honest parties agree on a value, the universal exchange before the second run makes sure all honest parties stick to it and the second run cannot change the agreed-upon value.

A tricky situation this universal exchange step needs to handle is when some honest parties stick to a value but other parties do not. In this case, this step needs to ensure that, if any honest party sticks to a value, other parties at least input that value to the subsequent BA call. The validity property of a correct recursive BA call will ensure agreement.

Here, the above recursive construction itself is independent of f, but the universal exchange step of Berman et al. relies on a quorum-intersection argument which only works under $f < n/3$. To make the framework independent of f, we abstract the functionality of this step as *graded Byzantine agreement* (GBA), since it is essentially the agreement version of graded broacast [16, 19]. In the rest of this section, we formally define the GBA primitive and construct a BA protocol using a GBA protocol as a black-box and prove its correctness.

3.1 Graded Byzantine Agreement

In graded Byzantine agreement (GBA), each party r has an input, and outputs a tuple (v, g) where v is the output value and $g \in \{0, 1\}$ is a grade bit.

▶ **Definition 8** (Graded Byzantine Agreement (GBA)). *A Graded Byzantine agreement protocol must satisfy the following properties.*

1. *consistency: if an honest party outputs $(v, 1)$, then all honest parties output $(v, *)$.*
2. *validity: if all honest parties have the same input value v, then all honest parties output $(v, 1)$*
3. *termination: every honest party outputs and terminates.*

The "stick to" nature is expressed by the grade bit g. The consistency property requires that if an honest party sticks to a value v, i.e., output v with $g = 1$, then all honest parties output the same value v. The validity property states that if all honest parties have the same input value v, they all stick to the value. These two properties capture what the universal exchange step needs to achieve explained at an intuitive level.

3.2 Recursive Construction of Byzantine Agreement

Next, we present the recursive BA protocol RBA in Figure 1. Let Q_w denote a set of parties that run a BA protocol. Since the protocol is recursive, the set Q_w is also defined recursively. Q_1 is a set of all n parties. Q_{2w} is the first $\lceil |Q_w|/2 \rceil$ parties in Q_w, and Q_{2w+1} is the remaining $\lfloor |Q_w|/2 \rfloor$ parties. All parties start by running RBA(Q_1) at the beginning.

RBA(Q_w)

Let r be a party. A variable v_r is initialized with the input value.

If $|Q_w| \leq M$, parties in Q_w execute any inefficient BA protocol with the $f < n/2$ resilience and output the decision of this inefficient BA. Otherwise, run the following steps, and output v_r and terminate.

1. Run GBA(Q_w) with input v_r. Let (v_r, g_r) be r's output from GBA(Q_w).
2. If $r \in Q_{2w}$, run RBA(Q_{2w}) with input v_r, and send the output v to all parties in Q_w. Otherwise, wait until the step finish.
3. If r receives the same value v from a majority of parties in Q_{2w} and $g_r = 0$, set v_r to v.

4. Run GBA(Q_w) with input v_r. Let (v_r, g_r) be r's output from GBA(Q_w).
5. If $r \in Q_{2w+1}$, run RBA(Q_{2w+1}) with input v_r, and send the output v to all parties in Q_w. Otherwise, wait until the step finish.
6. If r receives the same value v from a majority of parties in Q_{2w+1} and $g_r = 0$, set v_r to v.

Figure 1 Byzantine Agreement with $O(\kappa n^2)$ communication and $f < \frac{n}{2}$.

If the size of the RBA instance gets below a constant, denoted as M in the figure, parties can run any inefficient BA protocol with cubic or even higher communication complexity but with the desired resilience up to $f < n/2$. There are many such constructions in the literature [21, 14, 19, 2]; we do not describe these protocols. Otherwise, parties run two instances of RBA recursively to further reduce the instance size. Before each recursive call, they run a given GBA protocol denoted GBA. The grade bit output g_r of the GBA determines if a party r "sticks to" the GBA output or adopts the recursive RBA output.

Correctness of the Protocol. We prove the correctness of RBA for $f < n/2$ assuming the given GBA protocol GBA also tolerates $f < n/2$. The proof is easily extended for $f \leq (\frac{1}{2} - \varepsilon)n$. Below, minority faults within a set of parties Q mean at most $\lfloor(|Q| - 1)/2\rfloor$ faults.

▶ **Lemma 9.** RBA *solves BA in the presence of minority faults.*

Proof. Termination is obvious. The proof for validity is also easy. If all honest parties have the same input value $v_r = v$, then due to the validity of GBA, all honest parties output $(v, 1)$ in step-1. Thus, they do not change v_r at step-3 and input $v_r = v$ into the GBA of step-4. Again due to the validity of GBA, all honest parties output $(v, 1)$ in step-4, do not change v_r at step-6, and all output v.

Next, we prove consistency. When $|Q| \leq M$, the correctness of RBA reduces to the correctness of the given inefficient BA. We just need to prove for the recursive step. Specifically, we will prove that RBA solves BA under n parties with minority faults, if RBA solves BA under $< n$ parties with minority faults.

Consider RBA(Q_w). Since Q_w has minority faults, at least one of the two halves Q_{2w} and Q_{2w+1} has minority faults. Let us first consider the case where Q_{2w} has minority faults. Here, there are two situations with regard to the result of step-1: (i) all honest parties in Q_w set g_r to 0, or (ii) at least an honest party in Q_w sets g_r to 1.

In the first situation, all honest parties will set v_r to the majority output of step-2. By the consistency and termination of $\mathsf{RBA}(Q_{2w})$, all honest parties in Q_w receive the same value v from honest parties in Q_{2w} (which constitute a majority in Q_{2w}). Thus, all honest parties in Q_w set v_r to v in step-3.

In the second situation, since some honest party sets g_r to 1 in step-1, then by the consistency of GBA, all honest parties in Q_w set v_r to the same value v at the end of step-1. By the validity of $\mathsf{RBA}(Q_{2w})$, all honest parties in Q_{2w} output v, so all honest parties in Q_w receive v from a majority of parties in Q_{2w}. Thus, all honest parties in Q_w set v_r to v in step-3.

Therefore, in both situations, all honest parties in Q_w have the same value $v_r = v$ at the beginning of step-4. Then, by the validity of GBA, all honest parties in Q_w set (v_r, g_r) to $(v, 1)$ in step-4, so will not change their v_r in step-6 and all output the same value v.

The other case where Q_{2w+1} has minority faults can be proved similarly. No matter which of the two situations holds at step-4 (all have $g_r = 0$ or some have $g_r = 1$), all honest parities in Q_w have the same value $v_r = v$ at the end of step-6 and output the same value v. Therefore, regardless of whether Q_{2w} or Q_{2w+1} has minority faults, consistency holds. ◀

With some foresight, we will construct GBA with quadratic communication in the later section. This will give RBA with quadratic communication in total.

▶ **Lemma 10** (Communication Complexity). *If the communication complexity of* GBA *is* $O(\kappa n^2)$*, then the communication complexity of* RBA *is* $O(\kappa n^2)$.

Proof. The communication complexity of RBA is given as a recurrence below. Let s be the number of parties in an RBA instance.

$$C(s) = \begin{cases} O(\kappa) & \text{(if } s \leq M) \\ C(\lfloor s/2 \rfloor) + C(\lceil s/2 \rceil) + O(\kappa s^2) & \text{(otherwise)} \end{cases}$$

For any n, the depth of the recursion k satisfies $2^{k-1}M \leq n \leq 2^k M$. Hence, $C(n) \leq 2^k O(\kappa)$ + $\sum_{i=0}^{k} 2^i O(\kappa(n/2^i)^2) = O(\kappa n^2)$. ◀

4 Two Constructions of Graded Byzantine Agreement

This section presents two constructions of GBA protocols with different trade-offs to instantiate two BA protocols from the recursive framework in the previous section and complete the proof of Theorem 1 and 2.

4.1 Graded Byzantine Agreement with Threshold Signature Scheme

We first present a GBA protocol (denoted $\frac{1}{2}$-GBA) with quadratic communication and $f < n/2$ assuming a threshold signature scheme, which complete the proof of Theorem 1. We describe $\frac{1}{2}$-GBA in Figure 2. The parameter Q is a set of parties that participate in the protocol. Let $n = |Q|$.

Intuitive overviews. The construction is inspired by a few recent work on synchronous BB and BFT protocols [2, 3, 5]. Rounds 1–3 form a set of $n - f$ vote-1 (vote1-certificate) for the same value v, denoted $\mathcal{C}^1(v)$. Here, if an honest party votes for a value v in round 3, it must have received and multicast $n - f$ echo (echo-certificate) for v, denoted $\mathcal{E}(v)$ in round 2. Moreover, if a party receives a conflicting echo-certificate $\mathcal{E}(v')$ by the end of round 2, it does not vote in round 3. Therefore, rounds 1 and 2 prevent conflicting vote1-certificates from being created.

$\frac{1}{2}$-GBA(Q)

Let r be a party. $n = |Q|$, and $f < \lfloor (n-1)/2 \rfloor$. A variable v_r is initialized to the input value. g is initialized to 0. Run the following within the set of parties Q. $\langle x \rangle_r$ is a signature share on message x of a $(n-f, |Q|)$-threshold signature.

1. Multicasts $\langle \mathsf{echo}, v_r \rangle_r$.
2. If r receives $n-f$ $\langle \mathsf{echo}, v \rangle_*$, combine them into a threshold signature denoted $\mathcal{E}(v)$, and then multicasts $\mathcal{E}(v)$.
3. If r have multicast $\mathcal{E}(v)$ in round 2, and does not receive $\mathcal{E}(v')$ $(v' \neq v)$ by the end of round 2, multicasts $\langle \mathsf{vote\text{-}1}, v \rangle_r$.
4. If r receives $n-f$ $\langle \mathsf{vote\text{-}1}, v \rangle_*$, combine them into a threshold signature denoted $\mathcal{C}^1(v)$, and then multicasts $\mathcal{C}^1(v)$ and $\langle \mathsf{vote\text{-}2}, v \rangle_r$.
   At the end of the round, if r receives $\mathcal{C}^1(v)$, sets v_r to v. If r receives $n-f$ $\langle \mathsf{vote\text{-}2}, v \rangle_*$, denoted $\mathcal{C}^2(v)$, sets g to 1.

Finally, outputs (v_r, g).

Figure 2 Graded Byzantine agreement with $f < n/2$ with a threshold signature scheme.

Round 4 forms a set of $n-f$ vote-2 (vote2-certificate) for a value v, denoted $\mathcal{C}^2(v)$. If a party receives a vote1-certificate $\mathcal{C}^1(v)$ by the end of round 3, it sends vote-2 for a value v (along with $\mathcal{C}^1(v)$) in round 4. Therefore, if a vote2-certificate $\mathcal{C}^2(v)$ is formed, all honest parties can receive a vote1-certificate $\mathcal{C}^1(v)$.

Finally, a party outputs a value v if it receives a vote1-certificate $\mathcal{C}^1(v)$, and it further sets the grade bit g to 1 if it also receives a vote2-certificate $\mathcal{C}^2(v)$. Consistency follows from the properties above. Moreover, if all honest parties have the same input value v, all honest parties (at least $n-f$) receive both $\mathcal{C}^1(v)$ and $\mathcal{C}^2(v)$ and output $(v, 1)$, so validity also holds.

Correctness of the protocol. We prove the correctness of $\frac{1}{2}$-GBA assuming $f < n/2$. The termination of $\frac{1}{2}$-GBA is trivial, and thus we prove the consistency and validity.

▶ **Lemma 11.** *If $\mathcal{C}^1(v)$ and $\mathcal{C}^1(v')$ are both created, then $v = v'$.*

Proof. Suppose $\mathcal{C}^1(v)$ is created, then at least an honest party r must have multicast vote-2 for v in round 3. That implies r received $\mathcal{E}(v)$ and multicast it in round 2. Then, all honest parties must have received $\mathcal{E}(v)$ by round 3, and all honest parties could not have multicast vote-2 for $v' \neq v$. Therefore, $\mathcal{C}^1(v')$ cannot be created unless $v' = v$. ◀

▶ **Lemma 12** (Consistency). *If an honest party outputs $(v, 1)$, then all honest parties output $(v, *)$*

Proof. Suppose an honest party outputs $(v, 1)$, then it must have received $\mathcal{C}^2(v)$ for a value v by the end of round 4. Then, at least one honest party must have multicast $\mathcal{C}^1(v)$ in round 4, and all honest parties must have received it by the end of round 4. Since there is not $\mathcal{C}^1(v')$ for a different value v' by Lemma 11, all honest parties set v_r to v at the end of round 4 and thus output v. ◀

▶ **Lemma 13** (Validity). *If all honest parties have the same input value v, then all honest parties output $(v, 1)$.*

Proof. If all honest parties have the same input value v, they all multicast $\langle \mathsf{echo}, v\rangle$ in round 1, and thus $\mathcal{E}(v)$ should be formed and $\mathcal{E}(v')$ for $v' \neq v$ cannot be formed. In the same way, all honest parties multicast $\langle \mathsf{vote\text{-}1}, v\rangle$ in round 3 and $\langle \mathsf{vote\text{-}2}, v\rangle$ in round 4. Therefore, $\mathcal{C}^1(v)$ and $\mathcal{C}^2(v)$ should be formed and $\mathcal{C}^1(v')$ and $\mathcal{C}^2(v')$ for $v' \neq v$ cannot be formed. Thus, all honest parties output $(v, 1)$. ◀

Communication complexity and discussion. With threshold signatures, all certificates $\mathcal{E}(v)$, $\mathcal{C}^1(v)$, $\mathcal{C}^2(v)$ are $O(\kappa)$ in size, and the commutation complexity of $\frac{1}{2}$-GBA is clearly $O(\kappa n^2)$. But we note that the RBA protocol in Figure 1 invokes GBA with different numbers of participants for each depth in the recursion and hence requires different thresholds for threshold signatures. As a result, each party needs $\Theta(\log n)$ key setups.

4.2 Graded Byzantine Agreement without Threshold Signature Scheme

Next, we present a GBA protocol (denoted $(\frac{1}{2} - \varepsilon)$-GBA) with quadratic communication and $f \leq (\frac{1}{2} - \varepsilon)n$ for any positive constant ε without relying on any threshold signature scheme or trusted setup (beyond the standard PKI). We describe $(\frac{1}{2} - \varepsilon)$-GBA in Figure 3.

Intuitive overview. The main motivation of $(\frac{1}{2} - \varepsilon)$-GBA is to remove the use of threshold signature. Thus, let us first review why threshold signature scheme is necessary in the GBA protocol from the previous section. The threshold signature scheme is used to aggregate a set of $n - f$ signatures (quorum certificate $\mathcal{E}(v)$ in round 2 and vote1-certificate $\mathcal{C}^1(v)$ in round 4). If these are not aggregated, each party needs to multicast linear-sized certificates, leading to cubic communication in total.

Therefore, to remove aggregation while keeping the communication quadratic, we need to remove multicast. However, multicasting quorum certificates in round 2 and 4 is key to consistency. Specifically, multicasting an echo-certificate $\mathcal{E}(v)$ in round 2 helps honest parties detect a conflicting echo-certificate $\mathcal{E}(v')$, which allows honest parties to decide the value v safely; multicasting a vote1-certificate $\mathcal{C}^1(v)$ in round 4 helps notify all honest parties of the existence of $\mathcal{C}^1(v)$, which allows the party to decide the value v with confidence, i.e., grade bit $g = 1$.

Our key new technique is to replace the multicast steps with more efficient yet robust dissemination of certificates through a predetermined expander graph with a constant degree.

▶ **Definition 14** (Expander). *An (n, α, β)-expander $(0 < \alpha < \beta < 1)$ is a graph of n vertices such that, for any set S of αn vertices, the number of neighbors of S is more than βn.*

It is well-known that for any n and $0 < \alpha < \beta < 1$, (n, α, β)-expanders exist. For our purpose, we need an $(n, 2\varepsilon, 1 - 2\varepsilon)$-expander; in other words, we set $\alpha = 2\varepsilon$ and $\beta = 1 - 2\varepsilon$. Henceforth, we write an $(n, 2\varepsilon, 1 - 2\varepsilon)$-expander as $G_{n,\varepsilon}$. For completeness, we show in Appendix A that for all positive ε and for all n, the required expander $G_{n,\varepsilon}$ always exists.

Instead of sending a quorum certificate to all other parties, a party propagates it to a constant number of neighbors in $G_{n,\varepsilon}$. Therefore, the total number of messages is reduced from quadratic to linear, and thus the total communication is kept quadratic even though some messages contain a linear number of signatures. Our key observation is that although the message is not sent to everyone (since the expander is not a fully connected graph), it is sufficient to maintain consistent decisions among honest parties.

In more detail, in round 3, each party multicasts $\mathsf{vote\text{-}1}$ for a value v only if it propagated an echo-certificate $\mathcal{E}(v)$ in round 2 and it does not receive a conflicting echo-certificate $\mathcal{E}(v')$. If a vote1-certificate $\mathcal{C}^1(v)$ forms, at least $n - 2f = 2\varepsilon n$ are honest. They must have

$(\frac{1}{2} - \varepsilon)$-GBA$(Q)$

Let r be a party. $n = |Q|$, and $f = \lfloor(\frac{1}{2} - \varepsilon)n\rfloor$. A variable v_r is initialized to the input value. g is initialized to 0. "Propagate" means sending to all neighbors in $G_{n,\varepsilon}$ and "multicast" means sending to all n parties. Run the following within the set of parties Q. $\langle x\rangle_r$ is a digital signature on a message x.

1. Multicasts $\langle \mathsf{echo}, v_r\rangle_r$.
2. If r receives $n - f$ $\langle \mathsf{echo}, v\rangle_*$, denoted $\mathcal{E}(v)$, propagates $\mathcal{E}(v)$.
3. If r have propagated $\mathcal{E}(v)$ in round 2, and does not receive $\mathcal{E}(v')$ $(v' \neq v)$ by the end of round 2, multicasts $\langle \mathsf{vote\text{-}1}, v\rangle_r$.
4. If r receives $n - f$ $\langle \mathsf{vote\text{-}1}, v\rangle_*$, denoted $\mathcal{C}^1(v)$, propagate $\mathcal{C}^1(v)$, and multicasts $\langle \mathsf{vote\text{-}2}, v\rangle_r$. .
5. If r receives $\mathcal{C}^1(v)$ by the end of round 4, multicasts $\langle \mathsf{vote\text{-}3}, v\rangle_r$.
   At the end of the round, if r receives $f + 1$ $\langle \mathsf{vote\text{-}3}, v\rangle_*$, sets v_r to v. If r receives $n - f$ $\langle \mathsf{vote\text{-}2}, v\rangle_*$, denoted $\mathcal{C}^2(v)$, set g to 1.

Finally, outputs (v_r, g).

Figure 3 Graded Byzantine agreement with $f \leq (\frac{1}{2} - \varepsilon)n$ without threshold signature scheme.

propagated $\mathcal{E}(v)$ and it will be received by more than $(1 - 2\varepsilon)n = 2f$ parties. Out of these, at least $f + 1$ are honest and will not vote for a conflicting value. This guarantee the unique existence of vote1-certificate $\mathcal{C}^1(v)$.

Confirming the existence of a vote1-certificate is trickier as we cannot afford multicasts to notify all parties. We achieve this in two steps. In round 4, after propagating $\mathcal{C}^1(v)$, the party multicast vote-2 for v. If a vote2-certificate $\mathcal{C}^2(v)$ forms, due to the expansion property, at least $f + 1$ honest parties receive $\mathcal{C}^1(v)$ by the end of round 4. Then, in round 5, if a party receives $\mathcal{C}^1(v)$, it multicast vote-3 message for v. As at least $f + 1$ honest parties receives $\mathcal{C}^1(v)$, all honest parties can receive $f + 1$ vote-3 message for v, which works as a succinct proof of existence of $\mathcal{C}^1(v)$. This allows all honest parties to confirm the existence of a vote1-certificate.

Correctness of the protocol. We prove the correctness of $(\frac{1}{2} - \varepsilon)$-GBA assuming $f \leq (\frac{1}{2} - \varepsilon)n$ for any positive constant ε. The termination of $(\frac{1}{2} - \varepsilon)$-GBA is trivial, and thus we prove the consistency and validity.

▶ **Lemma 15.** *If $\mathcal{C}^1(v)$ and $\mathcal{C}^1(v')$ are both created, then $v = v'$.*

Proof. Suppose $\mathcal{C}^1(v)$ is created, then at least $2\varepsilon n$ honest parties must have propagated $\mathcal{E}(v)$ in round 2. Then, due to the expansion property of $G_{n,\varepsilon}$, more than $2f$ parties, out of which at least $f + 1$ honest parties must have received $\mathcal{E}(v)$ by the end of round 2, and do not send $\langle \mathsf{vote\text{-}1}, v'\rangle_*$ for a different value $v' \neq v$ in round 3. Therefore, $\mathcal{C}^1(v')$ cannot be created unless $v' = v$. ◀

▶ **Lemma 16** (Consistency). *If an honest party outputs $(v, 1)$, then all honest parties output $(v, *)$.*

Proof. Suppose an honest party outputs $(v, 1)$, then it must have received $\mathcal{C}^2(v)$ for a value v by the end of round 5. Then, at least $2\varepsilon n$ honest parties must have propagated $\mathcal{C}^1(v)$ in round 4. Due to the expansion property of $G_{n,\varepsilon}$, more than $2f$ parties, out of which at least $f + 1$ honest parties must have received $\mathcal{C}^1(v)$ by the end of round 4, and multicast $\langle \mathsf{vote\text{-}3}, v\rangle_*$ in round 5. Thus, all honest parties must have received $f + 1$ $\langle \mathsf{vote\text{-}3}, v\rangle_*$ by the end of round 5. Here, as $\mathcal{C}^1(v')$ for a different value $v' \neq v$ cannot form by Lemma 15, honest parties could not have multicast $\langle \mathsf{vote\text{-}3}, v'\rangle_*$. Therefore, all honest party could not have received $f + 1$ $\langle \mathsf{vote\text{-}3}, v'\rangle_*$, and thus output v. ◀

▶ **Lemma 17** (Validity). *If all honest parties have the same input value v, then all honest parties output $(v, 1)$*

Proof. If all honest parties have the same input value v, they all multicast $\langle \mathsf{echo}, v\rangle$ in round 1, and thus $\mathcal{E}(v)$ must form and $\mathcal{E}(v')$ for $v' \neq v$ cannot form. Then, all honest parties multicast $\langle \mathsf{vote\text{-}1}, v\rangle$ in round 3, propagate $\mathcal{C}^1(v)$ and multicast $\langle \mathsf{vote\text{-}2}, v\rangle$ in round 4, and $\langle \mathsf{vote\text{-}3}, v\rangle$ in round 5. Therefore, all honest parties receive both $\mathcal{C}^2(v)$ and $f + 1$ $\langle \mathsf{vote\text{-}3}, v\rangle_*$, and output $(v, 1)$. ◀

Communication complexity. All certificates $\mathcal{E}(v)$, $\mathcal{C}^1(v)$, $\mathcal{C}^2(v)$ are $O(\kappa n)$ in size, but are only sent through the degree-d expander. All the multicasted messages are $O(\kappa)$ in size. Thus, the communication complexity of $(\frac{1}{2} - \varepsilon)$-GBA is $O(\kappa n^2 d)$. Appendix A shows that $d = O(\frac{1}{\varepsilon})$ suffices, so the communication complexity of $(\frac{1}{2} - \varepsilon)$-GBA is $O(\kappa n^2)$ when ε is a constant, and is $O(\kappa n^2/\varepsilon)$ in general. This communication complexity is inherited in the RBA protocol in Figure 1.

5 Conclusion

In this paper, we provided two results: (1) a BA protocol with quadratic communication with optimal resilience $f < n/2$ with a trusted setup, and (2) a BA protocol with quadratic communication with near optimal resilience $f \leq (\frac{1}{2} - \varepsilon)n$ without trusted setup. Even with our new results, the tightness of the Dolev-Reischuk lower bound is still open for some settings, for exmaple, BA under a standard PKI model with $(\frac{1}{2} - \varepsilon)n < f < n/2$, or quadratic BB with $f \geq n/2$ even with a trusted setup. These are intriguing open questions for future work.

References

1 Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 317–326, 2019.

2 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous byzantine agreement with expected $o(1)$ rounds, expected $o(n^2)$ communication, and optimal resilience. In *Financial Cryptography and Data Security (FC)*, pages 320–334. Springer, 2019.

3 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *IEEE Symposium on Security and Privacy (S&P)*, pages 106–118. IEEE, 2020.

4 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Validated asynchronous byzantine agreement with optimal resilience and asymptotically optimal time and word communication. *arXiv preprint*, 2018. arXiv:1811.01332.

5 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Optimal good-case latency for byzantine broadcast and state machine replication. *arXiv preprint*, 2020. arXiv:2003.13155.

6 Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit optimal distributed consensus. In *Computer science*, pages 313–321. Springer, 1992.

7 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference (CRYPTO)*, pages 524–541. Springer, 2001.

8 Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 191–201. IEEE, 2005.

9 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186. USENIX, 1999.

10 Jing Chen and Silvio Micali. Algorand. *arXiv preprint*, 2016. arXiv:1607.01341.

11 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. *arXiv preprint*, 2020. arXiv:2002.06545.

12 Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

13 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

14 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

15 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

16 Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 148–161, 1988.

17 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

18 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

19 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.

20 Valerie King and Jared Saia. Breaking the $o(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.

21 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

22 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2020.

23 Silvio Micali. Byzantine agreement, made trivial, 2016.

24 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *ACM Conference on Computer and Communications Security (CCS)*, pages 31–42, 2016.

25 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. *arXiv preprint*, 2020. arXiv:2002.11321.

26 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

27 Alexander Spiegelman. In search for a linear byzantine agreement. *arXiv preprint*, 2020. arXiv:2002.06993.

28 Georgios Tsimos, Julian Loss, and Charalampos Papamanthou. Nearly quadratic broadcast without trusted setup under dishonest majority. *IACR Cryptology ePrint Archive, Report 2020/894*, 2020.

29 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

A Expander

We show that an expander G_ε in the Definition 14 exists for all positive constant ε. We use $\Gamma(V, G)$ to denote a set of all neighbors of V in a graph G.

▶ **Theorem 18** (Existence of Expander). *For all positive integer n and positive ε, there exists an expander $G_{n,\varepsilon}$ with degree $d = O(\frac{1}{\varepsilon})$.*

Proof. Let $c = 2\varepsilon$ and $G_{n,\varepsilon}$ is an $(n, c, 1-c)$-expander. For $c \geq 1/2$, the expansion property becomes trivial. Note that for our purpose, we can consider a vertex a neighbor of itself, so a graph with a self edge for every vertex is an $(n, c, 1-c)$-expander for $c \geq 1/2$. So we just need to focus on $c < 1/2$.

Consider a random d degree graph G taking the union of random d perfect matchings (if n is odd, the first party has two links). In each perfect matching P, for any set of cn parties (say S), and any set of $(1-c)n$ parties (say T), the probability that $\Gamma(S, P) \subseteq T$ is at most

$$\Pr[\Gamma(S, P) \subseteq T] \leq \left(\frac{(1-c)n}{n}\right)^{\frac{cn}{2}} = (1-c)^{\frac{cn}{2}}.$$

Thus, the probability that any set of cn parties do not expand in the graph, i.e., $|\Gamma(S, G)| \leq (1-c)n$ for any S, is at most

$$\begin{aligned} & \binom{n}{cn}\binom{n}{(1-c)n}(1-c)^{\frac{cdn}{2}} \\ \leq\ & \left(\frac{e}{c}\right)^{cn}\left(\frac{e}{1-c}\right)^{(1-c)n}(1-c)^{\frac{cdn}{2}} \\ \leq\ & \left(e\left(\frac{1}{c}\right)^{c}\left(\frac{1}{1-c}\right)^{1-c}(1-c)^{\frac{cd}{2}}\right)^{n} \end{aligned}$$

The above probability upper bound is smaller than 1 (in fact, exponentially small in n), when the degree d is sufficiently large. The precise requirement on d is $\frac{d}{2} > \frac{1}{c} - 1 + \frac{c \log c - \log e}{c \log(1-c)}$. It is not hard to show that when $0 < c < \frac{1}{2}$, $d = O(\frac{1}{c}) = O(\frac{1}{\varepsilon})$ will suffice. This means there is a non-zero (in fact, overwhelmingly large) probability that a randomly chosen graph is an expander. Thus, $G_{n,\varepsilon}$ with degree $d = O(\frac{1}{\varepsilon})$ exists. ◀

Algorithms for the Minimum Dominating Set Problem in Bounded Arboricity Graphs: Simpler, Faster, and Combinatorial

Adir Morgan
Tel Aviv University, Israel

Shay Solomon
Tel Aviv University, Israel

Nicole Wein
MIT, Cambridge, MA, USA

Abstract

We revisit the minimum dominating set problem on graphs with arboricity bounded by α. In the (standard) centralized setting, Bansal and Umboh [6] gave an $O(\alpha)$-approximation LP rounding algorithm, which also translates into a near-linear time algorithm using general-purpose approximation results for explicit mixed packing and covering or pure covering LPs [39, 57, 1, 50]. Moreover, [6] showed that it is NP-hard to achieve an asymptotic improvement for the approximation factor. On the other hand, the previous two *non*-LP-based algorithms, by Lenzen and Wattenhofer [43], and Jones et al. [36], achieve an approximation factor of $O(\alpha^2)$ in *linear* time.

There is a similar situation in the distributed setting: While there is an $O(\log^2 n)$-round LP-based $O(\alpha)$-approximation algorithm implied in [40], the best non-LP-based algorithm by Lenzen and Wattenhofer [43] is an implementation of their centralized algorithm, providing an $O(\alpha^2)$-approximation within $O(\log n)$ rounds.

We address the questions of whether one can achieve an $O(\alpha)$-approximation algorithm that is *elementary*, i.e., not based on any LP-based methods, either in the centralized setting or in the distributed setting. We resolve both questions in the affirmative, and en route achieve algorithms that are faster than the state-of-the-art LP-based algorithms. Our contribution is two-fold:

1. In the centralized setting, we provide a surprisingly simple combinatorial algorithm that is asymptotically optimal in terms of both approximation factor and running time: an $O(\alpha)$-approximation in *linear* time. The previous state-of-the-art $O(\alpha)$-approximation algorithms are (1) LP-based, (2) more complicated, and (3) have super-linear running time.
2. Based on our centralized algorithm, we design a distributed combinatorial $O(\alpha)$-approximation algorithm in the CONGEST model that runs in $O(\alpha \log n)$ rounds with high probability. Not only does this result provide the first nontrivial *non*-LP-based distributed $o(\alpha^2)$-approximation algorithm for this problem, it also outperforms the best LP-based distributed algorithm for a wide range of parameters.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Graph Algorithms, Dominating Set, Bounded Arboricity, Linear time algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.33

Related Version This paper is an abbreviated version of the following full paper:
Full Version: https://arxiv.org/pdf/2102.10077.pdf

Funding *Shay Solomon*: Partially supported by the Israel Science Foundation grant No.1991/19.
Nicole Wein: Supported by NSF Grant CCF-1514339.

Acknowledgements We thank Yosi Hezi and Quanquan Liu for fruitful discussions. We would also like to thank Neal E. Young and Kent Quanrud for correspondence about their prior work.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 33; pp. 33:1–33:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Background

The minimum dominating set (MDS) problem is a classic combinatorial optimization problem. Given a graph G we want to find a minimum cardinality set D of vertices, such that every vertex of the graph is either in D or has a neighbor in D. Besides its theoretical implications, solving this basic problem efficiently has many practical applications in domains ranging from wireless networks to text summarizing (see, e.g., [56, 46, 52]). The MDS problem was one of the first problems recognized as NP-complete [27]. It was also one of the first problems for which an approximation algorithm was analyzed: a simple greedy algorithm achieves a $\ln n$-approximation in general graphs [35]. This approximation factor is optimal up to lower order terms unless $\mathsf{P} = \mathsf{NP}$ [19].

Distributed MDS in general graphs. The first efficient distributed approximation algorithm for MDS was given by Jia, Rajaraman, and Suel [34], who gave a randomized $O(\log \Delta)$-approximation in $O(\log^2 n)$ rounds in the CONGEST model. This was improved by Kuhn, Moscibroda and Wattenhofer [40], who gave a randomized $(1+\varepsilon)(1+\ln(\Delta+1))$-approximation in $O(\log^2 \Delta/\varepsilon^4)$ rounds in the CONGEST model and in $O(\log n/\varepsilon^2)$ rounds in the LOCAL model. Ghaffari, Kuhn, and Maus [30] showed that by allowing exponential-time local computation, one can get a randomized $(1 + o(1))$-approximation in a polylogarithmic number of rounds in the LOCAL model. This result was derandomized by the network decomposition result of Rozhoň and Ghaffari [51]. From the lower bounds side, Kuhn, Moscibroda, and Wattenhofer [41] showed that getting a polylogarithmic approximation ratio requires $\Omega(\sqrt{\frac{\log n}{\log\log n}})$ and $\Omega(\frac{\log \Delta}{\log\log \Delta})$ rounds in the LOCAL model.

For *deterministic* distributed algorithms, improving over previous work, Deurer, Kuhn, and Maus [17] recently gave two algorithms in the CONGEST model with approximation factor $(1+\varepsilon)\ln(\Delta+1)$ for $\varepsilon > 1/\text{polylog}\Delta$, running in $2^{O(\sqrt{\log n \log\log n})}$ and $O((\Delta+\log^* n)\text{polylog}\Delta)$ rounds, respectively; the running time of the former CONGEST algorithm [29], achieving approximation factor $O(\log^2 n)$, is dominated by the time needed for deterministically computing a network decomposition in the CONGEST model, which, due to [28], is thus reduced to $O(\text{poly}\log n)$.

Graphs of bounded arboricity. The MDS problem has been studied on a variety of restricted classes of graphs, such as graphs with bounded degree (e.g., [14]), planar and bounded genus graphs (e.g., [5, 16, 3]), and graphs of bounded arboricity – which is the focus of this paper. The class of bounded *arboricity* graphs is a wide family of *uniformly sparse* graphs, defined as follows:

▶ **Definition 1.** *Graph G has* arboricity *bounded by α if $\frac{m_s}{n_s-1} \leq \alpha$, for every $S \subseteq V$, where m_s and n_s are the number of edges and vertices in the subgraph induced by S, respectively.*

The class of bounded arboricity graphs contains the other graph classes mentioned above as well as bounded treewidth graphs, and in general all graphs excluding a fixed minor. Moreover, many natural and real world graphs, such as the world wide web graph, social networks and transaction networks, are believed to have bounded arboricity. Consequently, this class of graphs has been subject to extensive research, which led to many algorithms for bounded arboricity graphs in both the (classic) centralized setting (e.g. [23, 32, 13]) and in the distributed setting (e.g. [15, 7, 31, 54]); there are also many algorithms in other settings, such as dynamic graph algorithms, sublinear algorithms and streaming algorithms (see [11, 33, 49, 48, 47, 53, 37, 20, 21, 22, 9, 44, 10, 8], and the references therein).

In distributed settings, one cannot always assume that all processors know the arboricity of the graph, so it is important to devise robust algorithms, which can perform correctly also when the arboricity is unknown to the processors (see e.g. [7, 43]).

1.2 Approximating MDS on graphs of arboricity α

Centralized setting. In the centralized setting, there are two non-LP-based algorithms for MDS for graphs of arboricity (at most) α (for brevity, in what follows we may write graphs of "arboricity α" instead of arboricity *at most* α). One is by Lenzen and Wattenhofer [43], the other is by Jones, Lokshtanov, Ramanujan, Saurabh, and Suchý [36], and both achieve an $O(\alpha^2)$-approximation in deterministic linear time[1]. There is also a very simple LP rounding algorithm by Bansal and Umboh that gives a 3α-approximation [6]. This algorithm is very simple, after the LP has been solved. To solve the LP, there are near-linear time general-purpose approximation algorithms for explicit mixed packing and covering or pure covering LPs [39, 57, 1, 50]. Combining such an algorithm with [6] yields an $O(\alpha)$-approximation for MDS, either deterministically within $O(m \log n)$ time [57] or randomly (with high probability) within $O(n \log n + m)$ time [39]. The latter bound is super-linear in the entire (non-degenerate) regime of arboricity $\alpha = o(\log n)$; the regime $\alpha = \Omega(\log n)$ is considered degenerate, since in that case one can use the greedy linear-time $\ln n$-approximation algorithm. Bansal and Umboh [6] also proved that achieving asymptotically better approximation is NP-hard.[2]

Distributed setting. In the distributed setting, there are two non-LP-based algorithms for MDS for graphs of arboricity α, both by Lenzen and Wattenhofer [43]. The first is a randomized $O(\alpha^2)$-approximation algorithm in the CONGEST model that runs in $O(\log n)$ rounds with high probability. This algorithm was made deterministic by Amiri [2], and uses an LP-based subroutine of Even, Ghaffari, and Medina [24]. The second algorithm of Lenzen and Wattenhofer is a deterministic $O(\alpha \log \Delta)$-approximation algorithm in the CONGEST model that runs in $O(\log \Delta)$ rounds, where Δ is the maximum degree.

Regarding LP-based algorithms, Kuhn, Moscibroda, and Wattenhofer [40] developed a general-purpose method for solving LPs of a particular structure in the distributed setting. It seems that by applying their method (specifically, Corollary 4.1 of [40]) to the LP approximation result of Bansal and Umboh in bounded arboricity graphs [6], one can get a deterministic $O(\alpha)$-approximation algorithm for MDS in the CONGEST model that runs in $O(\log^2 \Delta)$ rounds, but such a result has not been explicitly claimed in the literature.

A natural question. The aforementioned results demonstrate a significant gap for MDS algorithms in bounded arboricity graphs when comparing LP-based methods to elementary *combinatorial* approaches. It is natural to ask whether this gap can be bridged.

- In the centralized setting, is there any efficient non-LP-based $O(\alpha)$-approximation algorithm for MDS (even one that is slower than the aforementioned $O(m \log n)$ time deterministic and $O(n \log n + m)$ time randomized LP-based algorithms)? Further, can one achieve an $O(\alpha)$-approximation in *linear time* using *any* (even LP-based) algorithm?

[1] Note that the theorem statement of [43] has a typo suggesting that the approximation factor is $O(\alpha)$.

[2] Achieving $(\alpha - 1 - \varepsilon)$-approximation is NP-hard for any $\varepsilon > 0$ and any fixed α; achieving $(\lfloor \alpha/2 \rfloor - \varepsilon)$-approximation is NP-hard for any $\varepsilon > 0$ and any $\alpha = 1, \ldots, \log^{\delta} n$, for some constant δ [6, 18].

- In the distributed setting, is there any efficient non-LP-based distributed $O(\alpha)$-approximation algorithm for MDS? Further, can one achieve an $O(\alpha)$-approximation in the CONGEST model within $o(\log^2 \Delta)$ rounds using *any* (even LP-based) algorithm?

We note the caveat that there is no clear-cut distinction between combinatorial and non-combinatorial algorithms, but we operate under the premise that an algorithm is combinatorial if all its intermediate computations have a natural combinatorial interpretation in terms of the original problem. While all algorithms presented in this paper are certainly combinatorial under this premise, it is far less clear whether prior work is. In particular, the previous state-of-the-art LP-based approaches are based on general-purpose primal/dual methods; when restricted to the MDS problem, it is possible that these methods could reduce, after proper adaptations, into simpler combinatorial algorithms. Nonetheless, even if possible, it is unlikely that the resulting algorithm would be as simple and elementary as ours. In the distributed setting, [42] gives an LP-based algorithm specifically for MDS that is simpler than the subsequent general-purpose LP-based algorithm of Moscibroda, and Wattenhofer [40]; however, [42] is inferior to [40] in both approximation ratio and running time.

1.3 Our Contributions

We answer all parts of the above question in the affirmative. In particular, we give algorithms that achieve the asymptotically optimal approximation factor of $O(\alpha)$, and are not only simple and elementary, but also run faster than all known algorithms, including LP-based algorithms. We note that $O(\alpha)$ is the asymptotically optimal approximation factor for polynomial time algorithms in the centralized setting and also in distributed settings where processors are assumed to have polynomially-bounded processing power.

Centralized Setting. Our core contribution is an asymptotically optimal algorithm in the centralized setting.

▶ **Theorem 2.** *For graphs of arboricity α, there is an $O(m)$ time $O(\alpha)$-approximation algorithm for MDS.*

We note that our algorithm works even when α is not known a priori, since there is a linear time 2-approximation algorithm for computing the arboricity of a graph [4].

Our algorithm is asymptotically optimal in both running time and approximation factor: it runs in linear time, and asymptotically improving the approximation factor it gets is proved to be NP-hard [6]. (The constant in the approximation ratio is not tight; our algorithm gives an 8α-approxmation.) While the quantitative improvement in running time over prior work is admittedly minor (a logarithmic factor over the deterministic algorithm, and $\log n/\alpha$ over the randomized algorithm), still getting a truly linear time algorithm is qualitatively very different than an almost-linear time. Indeed, the study of linear time algorithms has received much attention over the years, even when it comes to shaving factors that grow as slowly as inverse-Ackermann type functions. This line of work includes celebrated breakthroughs in computer science: For example, for the Union-Find data structure, efforts to achieve a linear time algorithm led to a lower bound showing that inverse-Ackermann function dependence is necessary [25], matching the upper bound [55], which is a cornerstone result in the field. Another example is MST, where the inverse-Ackermann function was shaved from the upper bound of [12] to achieve a linear time algorithm either using randomization [38] or when the edge weights are integers represented in binary [26], but it remains a major open problem whether or not there exists a linear time deterministic comparison-based MST algorithm.

Distributed Setting. We demonstrate the applicability of our centralized algorithm, by using its core ideas to develop a distributed algorithm.

▶ **Theorem 3.** *For graphs of arboricity α, there is a randomized distributed algorithm in the* CONGEST *model that gives an $O(\alpha)$-approximation for MDS and runs in $O(\alpha \log n)$ rounds. The bound on the number of rounds holds with high probability (and in expectation). The algorithm works even when either α or n is unknown to each processor.*

For the "interesting" parameter regime where Δ is polynomial in n, and $\alpha = o(\log n)$, the number of rounds in our algorithm beats the prior work obtained by combining [40] and [6] which appears to run in $O(\log^2 \Delta)$ rounds; as noted already, such an algorithm has not been claimed explicitly before. We note the caveat that our algorithm is randomized while their algorithm appears to be deterministic.

In the process of obtaining our distributed algorithm, we also obtain a *deterministic* algorithm in the LOCAL model (with polynomial message sizes) in a polylogarithmic number of rounds, via reduction to the maximal independent set (MIS) problem:

▶ **Theorem 4.** *Suppose there is a deterministic (resp., randomized) distributed algorithm in the* LOCAL *model for computing an MIS on a general graph in $R(n)$ rounds. Then, for graphs of arboricity α, there is a deterministic (resp., randomized) distributed algorithm in the* LOCAL *model that gives an $O(\alpha)$-approximation for MDS in $O(R(n) \cdot \alpha^2 \log n)$ rounds. The algorithm works even when either α or n is unknown to each processor.*

While Theorem 4 is the first deterministic non-LP-based algorithm to achieve an $O(\alpha)$-approximation, we note that the LP-based approach obtained by combining [40] and [6] appears to achieve fewer rounds and work in the CONGEST model. Theorem 4 is not our main result and is used as a stepping stone towards our $O(\alpha \log n)$ round algorithm in the CONGEST model, which is deferred to the full version [45] due to space constraints.

We finally note that unlike in the centralized setting, handling unknown α in the distributed setting it is not trivial and requires special treatment; in the full version [45] we demonstrate that all of our distributed algorithms can cope with unknown α without increasing the approximation factor and running time beyond constant factors.

Wider applicability. We have demonstrated the applicability of our centralized algorithm to the distributed setting. We anticipate that the core idea behind our centralized algorithm could be applied more broadly, to other settings that involve locality. Perhaps the prime example in this context is the standard (centralized) setting of dynamic graph algorithms, where the graph undergoes a sequence of edge updates (a single edge update per step), and the algorithm should maintain the graph structure of interest ($O(\alpha)$-approximate MDS in our case) with a small *update time* – preferably poly log(n) and ideally $O(1)$.

1.4 Technical overview

Centralized algorithm. As a starting point, we consider the algorithm of Jones, Lokshtanov, Ramanujan, Saurabh, and Suchỳ [36], which achieves an $O(\alpha^2)$-approximation in linear time. Their algorithm is as follows. They iteratively build a dominating set D and maintain a partition of the remaining vertices into the dominated vertices B (the vertices that have a neighbor in D), and the undominated vertices W. This partition of the vertices, as well as further partitioning described later, is shown in Figure 1. The basic property of arboricity α graphs used by their algorithm is that every subgraph contains a vertex of degree $O(\alpha)$. They begin by choosing a vertex v with degree $O(\alpha)$ and adding v along with

v's entire neighborhood $N(v)$ to D. The intuition behind this is that at least one vertex in $\{v\} \cup N(v)$ must be in OPT (an optimal dominating set), since OPT must dominate v. Hence, they add at least one vertex in OPT and use that to pay for adding $O(\alpha)$ vertices not in OPT. We say that a vertex w *witnesses* v and the vertices in $N(v)$ that are added to D, if $w \in OPT \cap (\{v\} \cup N(v))$. Now, the goal of the algorithm is to iteratively choose vertices v to add to D along with $O(\alpha)$ many of v's neighbors so that each vertex in OPT witnesses $O(\alpha)$ vertices v along with $O(\alpha)$ neighbors for each such vertex v. That is, each vertex in OPT witnesses $O(\alpha^2)$ vertices in D, which yields an $O(\alpha^2)$-approximation.

To choose which vertices v and which $O(\alpha)$ of v's neighbors to add to D, they partition the set B into two subsets B_{low} and B_{high}, which are the sets of vertices in B with low and high degree to W, respectively, where the degree threshold is $\delta\alpha$ for some constant δ. We also define $W_{low} \subseteq W$ (differently from the notation of [36]) as the subset of vertices with degree at most $\delta\alpha$ in the subgraph induced by $W \cup B_{high}$. They add a vertex $w \in W_{low}$ to D along with w's $O(\alpha)$ neighbors that are in $W \cup B_{high}$. In the interest of brevity, we will not motivate why this scheme achieves the desired outcome that each vertex in OPT witnesses $O(\alpha^2)$ vertices in D.

The key innovation in our algorithm that allows us to reduce the approximation factor from $O(\alpha^2)$ to $O(\alpha)$ is a simple but powerful idea. After choosing a vertex w to add to D, we do not *immediately* add $O(\alpha)$ of w's neighbors to D. Instead w casts a "vote" for these $O(\alpha)$ neighbors, and only once a vertex gets $\delta\alpha$ many votes is it added to D. With this modification, we can argue that each vertex in OPT still witnesses $O(\alpha)$ such vertices w as in the previous approach, but the catch here is that each such vertex w contributes only $O(1)$ neighbors to D on average, so each vertex in OPT only witnesses a *total* of $O(\alpha)$ vertices in D, rather than $O(\alpha^2)$. Moreover, it is trivial to implement this algorithm in linear time.

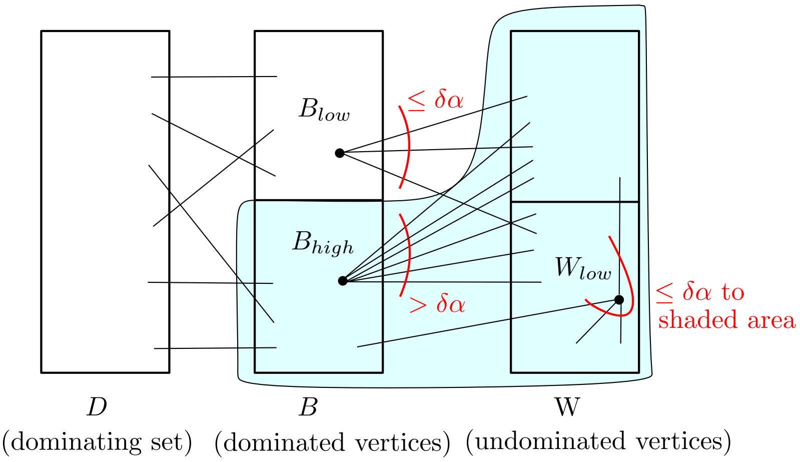


Figure 1 The partition of V into D, B and W, and further partitions of B and W.

Distributed algorithms using MIS. This section concerns the proof of Theorem 4: our reduction from MDS to MIS in the LOCAL model. This section also concerns a modification of this reduction that gives an $O(\alpha^2 \log^2 n)$ round algorithm in the CONGEST model. We use this algorithm as a stepping stone towards obtaining our main distributed algorithm (Theorem 3) which runs in $O(\alpha \log n)$ rounds in the CONGEST model.

We adapt our centralized algorithm to the distributed setting as follows. Recall that in our centralized algorithm, we repeatedly choose a vertex $w \in W_{low}$, add w to D, and cast a vote for each vertex in $N(w) \cap (W \cup B_{high})$. For our distributed algorithms, we would like to choose *many* such vertices w and process them in *parallel.* In fact, a constant fraction of the vertices in $W \cup B_{high}$ could be chosen as our vertex w since a constant fraction of vertices in a graph of arboricity α have degree $O(\alpha)$. However, we cannot simply process all of these vertices in parallel. In particular, if a vertex v has many neighbors being processed in parallel, v might accumulate many votes during a single round. This would invalidate the analysis of the algorithm, which relies on the fact that once a vertex v receives $\delta\alpha$ votes, v enters D.

To overcome this issue, we compute an MIS with respect to a 2-hop graph built from a subgraph of "candidate" vertices, and only process the vertices in this MIS in parallel. This MIS has two useful properties: 1. Its maximality implies that in any 2-hop neighborhood of a candidate vertex there is a vertex in the MIS; this helps to bound the number of rounds, and 2. Its independence implies that every vertex has at most one neighbor in the MIS, which ensures that any vertex can only receive one vote per round. To conclude, this approach gives a reduction from distributed MDS to distributed MIS in the LOCAL model. This approach can be made to work in the CONGEST model by replacing the black-box MIS algorithm with a 2-hop version of Luby's algorithm. This approach of running the 2-hop version of Luby's algorithm was also used in [43] for their distributed (α^2)-approximation for MDS.

Faster randomized distributed algorithm. In the CONGEST model, our distributed algorithm using MIS runs in $O(\alpha^2 \log^2 n)$ rounds with high probability. We devise a new, more nuanced algorithm that decreases the number of rounds to $O(\alpha \log n)$ with high probability. Our new algorithm is based on our previous algorithm, but with two key modifications, which save factors of $\log n$ and α, respectively.

Our first key modification, which shaves a $\log n$ factor from the number of rounds, is that we do not run an MIS algorithm as a black box. Instead, we run only a single phase of a Luby-like MIS algorithm before updating the data structures. Intuitively, this saves a $\log n$ factor because we are running just one phase of a $O(\log n)$-phase algorithm, but it is not clear a priori if we achieve the same progress as Luby's algorithm in a single phase. We show that this is indeed the case via more refined treatment of the behavior of each edge.

Our second key modification, which shaves an α factor from the number of rounds, concerns the Luby-like algorithm. Recall that in Luby's algorithm, each vertex v picks a random value $p(v)$ and then joins the MIS if $p(v)$ is the local minimum. In our algorithm, a vertex v instead joins the dominating set if $p(v)$ is an α-minimum, which roughly means that $p(v)$ is among the α smallest values that it is compared to. We show that with this relaxed definition, we still have the desired property that no vertex receives more than $\delta\alpha$ votes in a single round.

The main technical challenge is the analysis of the number of rounds. It is tempting to use an analysis similar to that of Luby's algorithm, where we count the expected number of "removed edges" over time. However, our above modifications introduce several complications that preclude such an analysis. Instead, we use a carefully chosen function to measure our progress. Throughout the algorithm, we add "weight" to particular edges, and our function measures the "total available weight". Specifically, whenever a vertex v is added to the dominating set, v adds *weight* to a particular set of edges in its 2-hop neighborhood. We show that the total amount of weight added in a single iteration of the algorithm decreases the total available weight substantially, which allows us to bound the total number of iterations.

All of our distributed algorithms so far have assumed that α is known to each processor but that n is unknown. We additionally show that all of them can be made to work in the setting where α is unknown but n is known. The idea of this modification is to guess

$\log n$ values of α and run a truncated version of the algorithm for each guess. However, it is impossible for an individual processor to know which guess of α is the most accurate without knowing the whole graph, so the processors cannot coordinate their guesses globally. We end up with different processors using different guesses of α, but we show that we can nonetheless obtain an algorithm whose approximation factor and running time are in accordance with the correct α.

1.5 Organization

Section 2 is for preliminaries. In Section 3, we present our centralized algorithm (Theorem 2). In Section 4, we present our distributed algorithms using MIS: in the LOCAL model we prove Theorem 4, and in the CONGEST model we give a randomized algorithm with $O(\alpha^2 \log^2 n)$ rounds, as a warm-up for the faster algorithm of Theorem 3. In section 5 of the full version [45] we prove Theorem 3 and that all our distributed algorithms can cope with unknown α.

2 Preliminaries

Let $G = (V, E)$ be an unweighted undirected graph. For any $S \subseteq V$, let $G[S]$ be denote the subgraph induced by S. For any $v \in V$, $N_G(v)$ denotes the neighborhood of v, and $\deg_G(v) = |N_G(v)|$ denotes the degree of v. When the graph G is clear from context, we omit the subscript.

Our distributed algorithm apply to the LOCAL and CONGEST models of distributed computing; definitions can be found in section 2 of the full version [45]. For the problem of MDS in both models, the requirement is that at the end of the computation, every vertex knows whether or not it belongs to the dominating set.

The following two simple claims about graphs of bounded arboricity will be useful.

▷ Claim 5. In a graph of arboricity α, every subgraph contains a vertex of degree $\leq 2\alpha$.

▷ Claim 6. In a graph G with arboricity α, at least half of the vertices in any subgraph have degree at most 4α.

3 Linear time $O(\alpha)$-approximation for MDS

In this section we will prove Theorem 2.

3.1 Algorithm

A description of our algorithm is as follows. See Algorithm 1 for the pseudocode.

We first introduce some notation. Since our algorithm builds off of [36], we stick to their notation for the most part. See Figure 1. We define a constant δ and let $\delta\alpha$ be our *degree threshold*. We will set $\delta = 2$, but we use the variable δ so that our analysis also applies to our distributed algorithms, where δ is a different constant. We maintain a partition of the vertices into three sets: D, B, and W, where initially $D = \emptyset$, $B = \emptyset$, and $W = V$. The set D is our current dominating set, the set B is the vertices not in D with at least one neighbor in D, and the set W is the remaining vertices, i.e. the undominated vertices. The set B is further partitioned into two sets based on the degree of each vertex to W. Let $B_{low} = \{v \in B : |N(v) \cap W| \leq \delta\alpha\}$ and let $B_{high} = B \setminus B_{low}$. Let $W_{low} = \{v \in W : |N(v) \cap (W \cup B_{high})| \leq \delta\alpha\}$ Also, each vertex v has a *counter* c_v initialized to 0. (The counter c_v counts the number of "votes" that v receives, for the notion of "votes" introduced in the technical overview.)

First we claim that while W is nonempty, W_{low} is also nonempty. By Claim 5, $G[W \cup B_{high}]$ contains a vertex v of degree at most 2α. Since $\delta = 2$, v cannot be in B_{high} by the definition of B_{high}, so v must be in W, and hence in W_{low}.

The algorithm proceeds as follows. While there still exists an undominated vertex (i.e. while $W \neq \emptyset$), we do the following. First, we pick an arbitrary vertex $w \in W_{low}$ (we showed that W_{low} is nonempty). Then, for all $v \in N(w) \cap (W \cup B_{high})$, we increment c_v, and if $c_v = \delta\alpha$, we add v to D. Then, we add w to D. Lastly, we update the sets B, ${}_{low}$, B_{high}, W, and W_{low} according to their definitions. This concludes the description of the algorithm.

Algorithm 1 Linear time $O(\alpha)$-approximation for MDS.

```
Initialize partition: $D \leftarrow \emptyset$, $B = \emptyset$, $B_{high} \leftarrow \emptyset$, $B_{low} \leftarrow \emptyset$, $W \leftarrow V$, $W_{low} = \{v \in V : \deg(v) \leq \delta\alpha\}$
Initialize counters: $\forall v \in V : c_v \leftarrow 0$
while $W \neq \phi$ do
    $w \leftarrow$ a vertex in $W_{low}$
    for all $v \in N(w) \cap (W \cup B_{high})$ do
        $c_v \leftarrow c_v + 1$
        if $c_v = \delta\alpha$ then
            $D \leftarrow D \cup v$
    $D \leftarrow D \cup w$
    // Bookkeeping to update partition:
    $B = \{v : N(v) \cap D \neq \emptyset\}$
    $B_{low} = \{v \in B : |N(v) \cap W| \leq \delta\alpha\}$
    $B_{high} = B \setminus B_{low}$
    $W = V \setminus (D \cup B)$
    $W_{low} = \{v \in W : |N(v) \cap (W \cup B_{high})| \leq \delta\alpha\}$
Return D
```

3.2 Analysis

First, we note that D is indeed a dominating set because the algorithm only terminates once the set W of vertices that are not dominated, is empty.

3.2.1 Approximation ratio analysis

Let OPT be an optimal MDS. We will prove that the set D returned by Algorithm 1 is of size at most $4\delta\alpha \cdot |OPT|$.

We first make the next claim about the behavior of the partition of vertices over time.

▷ Claim 7.

(1) No vertex can ever leave D.
(2) No vertex can ever enter W from another set.
(3) No vertex can ever leave B_{low}.

Proof. Item 1 is by definition. Item 2 follows from item 1 combined with the fact that W is defined as the set of vertices with no neighbors in D. Now we prove item 3. A vertex from B_{low} cannot enter W by item 2. A vertex from B_{low} cannot enter B_{high} since the degree partition of B is based on degree to W, and by item 2 the degree of any vertex to W can only decrease over time. A vertex from B_{low} cannot enter D because there are two ways a vertex can enter D: on Algorithm 1 a vertex can only enter D from $W \cup B_{high}$, and on Algorithm 1 a vertex can only enter D from W. ◁

To show that $|D| \leq 4\delta\alpha \cdot |OPT|$, we partition D into two sets, D_{active} and $D_{passive}$, and bound each of these sets separately. The set D_{active} consists of the vertices added to D due to being chosen as the vertex w; that is, the vertices added to D in Algorithm 1 of Algorithm 1. The set $D_{passive}$ consists of the vertices added to D as a result of their counters reaching $\delta\alpha$; that is, the vertices added to D in Algorithm 1 of Algorithm 1. We first bound $|D_{active}|$.

▷ Claim 8. $|D_{active}| \leq 2\delta\alpha \cdot |OPT|$.

Proof. For each vertex $v \in D_{active}$, we assign v to an arbitrary vertex $u \in N(v) \cap OPT$, and we say that u *witnesses* v. Such a vertex u exists since OPT is a dominating set. For each vertex $u \in OPT$, let $D_u \subseteq D_{active}$ be the set of vertices that u witnesses. Our goal is to show that for each $u \in OPT$, $|D_u| \leq 2\delta\alpha$.

Fix a vertex $u \in OPT$. We partition the vertices $v \in D_u$ into two sets $D_u[B_{low}]$ and $D_u[B_{high} \cup W]$. Let $D_u[B_{low}] \subseteq D_u$ be the vertices that enter D while u is in B_{low}. Let $D_u[B_{high} \cup W] \subseteq D_u$ be vertices that enter D while u is in $B_{high} \cup W$. We note that no vertex in D_u can enter D while u is in D, because by definition, every vertex in $D_{active} \supseteq D_u$ moves directly from W to D. Therefore, $D_u = D_u[B_{low}] \cup D_u[B_{high} \cup W]$.

We first bound $\left|D_u[B_{low}]\right|$. By definition, while u is in B_{low}, u has at most $\delta\alpha$ neighbors in W. Since no vertex can ever enter W by Claim 7, no vertex can ever enter $N(u) \cap W$. Therefore, starting from the time that u first enters B_{low}, the total number of vertices ever in $N(u) \cap W$ is at most $\delta\alpha$. Every vertex $v \in D_u[B_{low}]$ is in $N(u) \cap W$ right before moving to D, so $\left|D_u[B_{low}]\right| \leq \delta\alpha$. Next, we bound $D_u[B_{high} \cup W]$. By the specification of the algorithm, whenever a vertex $v \in D_u[B_{high} \cup W]$ enters D, the counter c_u is incremented. Once c_u reaches $\delta\alpha$, u is added to D. Therefore, $\left|D_u[B_{high} \cup W]\right| \leq \delta\alpha$.

Putting everything together, we have $|D_u| = \left|D_u[B_{low}]\right| + \left|D_u[B_{high} \cup W]\right| \leq 2\delta\alpha$. ◁

Now we bound $D_{passive}$.

▷ Claim 9. $|D_{passive}| \leq |D_{active}|$.

Proof. We will show that every vertex in $D_{passive}$ has at least $\delta\alpha$ neighbors in D_{active}, while every vertex in D_{active} has at most $\delta\alpha$ neighbors in $D_{passive}$. Then, by the pigeonhole principle, it follows that $|D_{passive}| \leq |D_{active}|$.

First, we will show that every vertex in $D_{passive}$ has at least $\delta\alpha$ neighbors in D_{active}. By definition, every vertex $v \in D_{passive}$ has had its counter c_v incremented $\delta\alpha$ times. Every time c_v is incremented, one of v's neighbors (the vertex w from Algorithm 1) is added to D, joining D_{active}. Each such neighbor of v that joins D_{active} is distinct since every vertex can be added to D at most once by Claim 7. Therefore, every vertex in $D_{passive}$ has at least $\delta\alpha$ neighbors in D_{active}.

Now we will show that every vertex in D_{active} has at most $\delta\alpha$ neighbors in $D_{passive}$. Fix a vertex $w \in D_{active}$. By definition, when w enters D, w is moved straight from W to D. Thus, by Claim 7, w is never in B. Therefore, w is added to D before any of its neighbors are added to D, as otherwise w would enter B. Therefore, when w enters D, all of w's neighbors that will enter $D_{passive}$ are in $B \cup W$. By Claim 7, no vertex in B_{low} can ever enter D, so actually, when w enters D all of w's neighbors that will enter $D_{passive}$ are in $B_{high} \cup W$. By definition, when w enters D, w has at most $\delta\alpha$ neighbors in $B_{high} \cup W$. Therefore, w has at most $\delta\alpha$ neighbors in $D_{passive}$. ◁

Combining Claim 8 and Claim 9, we have that $|D| = |D_{active}| + |D_{passive}| \leq 4\delta\alpha \cdot |OPT|$.

3.2.2 Running time analysis

Our goal is to prove that Algorithm 1 runs in $O(m)$ time.

Throughout the execution of the algorithm, we maintain a data structure that consists of the following:

- The partition of V into D, B, W; with subsets B_{low}, B_{high}, W_{low}
- The induced graph $G[W \cup B_{high}]$ represented as an adjacency list
- For each vertex $v \in W \cup B_{high}$, the quantities $|N(v) \cap W|$ and $|N(v) \cap (W \cup B_{high})|$

We can bound the time needed for maintaining the data structure using the following observations:

- The data structure can be initialized in $O(m)$ time
- To maintain this data structure, it suffices to scan the neighborhood of a vertex every time it move between subsets
- Every vertex moves between subsets a constant number times during the run of the algorithm
- Maintaining the data structure allows the algorithm to run in time $O(m)$

The first three observations implies that maintaining the data structure takes time $O(m)$. Together with the last observation, we have that the entire algorithm takes time $O(m)$. Full analysis and proof can be found in subsection 3.2.2 of the full version [45].

4 Distributed $O(\alpha)$-approximation for MDS using MIS

In this section we will prove Theorem 4. We also show how to modify of the proof of Theorem 4 to get a bound in the CONGEST model:

▶ **Theorem 10.** *For graphs of arboricity α, there is a randomized distributed algorithm in the* CONGEST *model that gives an $O(\alpha)$-approximation for MDS that runs in $O(\alpha^2 \log^2 n)$ rounds with high probability. The algorithm works even when either α or n is unknown to each processor.*

In the full version [45], we use the algorithm of Theorem 10 as a starting point to get an improved algorithm with $O(\alpha \log n)$ rounds.

The algorithms presented in this section assume that α is known to each processor but n is unknown. We defer discussion of handling unknown α to the full version [45].

4.1 Algorithm

4.1.1 Overview

Our algorithm is an adaptation of our centralized algorithm from Theorem 2 to the distributed setting. Recall that in our centralized algorithm, we repeatedly choose a vertex $w \in W_{low}$, add w to the dominating set, and increment the *counter* of w's neighbors that are in $W \cup B_{high}$. For our distributed algorithms, we would like process *many* vertices in W_{low} in parallel. There are in fact many vertices in W_{low} (if $\delta \geq 4$) since Claim 6 implies that at least half of the vertices in any subgraph has degree at most 4α. However, we cannot simply process all of W_{low} at once. In particular, if a vertex v has many neighbors being processed in parallel, v might have its counter incremented once for each of these neighbors. This is undesirable because the analysis of our centralized algorithm relies on the fact that once a vertex has its counter incremented to $\delta\alpha$, it is added to the dominating set. Therefore, we would like to guarantee that only a limited number of v's neighbors are processed in parallel.

This is where the MIS problem becomes relevant: we ensure that no vertex has more than one neighbor being processed in parallel by taking an MIS I with respect to the graph G_{low} defined as follows: the vertex set of G_{low} is W_{low}. There is an edge (u, v) in G_{low} if there is a path of length 2 between u and v in $G[W \cup B_{high}]$. Note that because no vertex has more than one neighbor in I, we can process all vertices in I in parallel and only increase the counter of each vertex by at most one.

The algorithms for Theorem 4 and Theorem 10 are identical except for the MIS subroutine. Theorem 4 is for the LOCAL model so we can simply run any distributed MIS algorithm that works in the LOCAL model on G_{low} as a black box. On the other hand, Theorem 10 is for the CONGEST model and because G_{low} can have higher degree than G, running an MIS algorithm directly on G_{low} could result in messages that become too large after translating the algorithm to run on G. To bypass this issue, we use a simple modification of Luby's algorithm that computes I using only small messages, without increasing the number of rounds.

4.1.2 Algorithm description

We provide a description of the algorithms here, and include the pseudocode in Algorithm 2. The only difference between the algorithms for Theorem 4 and Theorem 10 is the MIS subroutine, which we will handle separately later.

The sets D, B, W, B_{high}, B_{low}, and W_{low} are defined exactly the same as in our centralized algorithm, except we set $\delta = 4$ instead of $\delta = 2$ so that we can apply Claim 6 instead of Claim 5. We repeat the definitions here for completeness. The set D is our current dominating set, the set B is the vertices not in D with at least one neighbor in D, and the set W is the remaining vertices, i.e. the undominated vertices. The set B is further partitioned into two sets based on the degree of each vertex to W. Let $B_{low} = \{v \in B : |N(v) \cap W| \leq \delta\alpha\}$ and let $B_{high} = B \setminus B_{low}$. Also, let $W_{low} = \{v \in W : |N(v) \cap (W \cup B_{high})| \leq \delta\alpha\}$. Lastly, each vertex v has a *counter* c_v.

Each vertex v maintains the following information:

- The set(s) among D, B, W, B_{high}, B_{low}, and W_{low} that v is a member of.
- The quantity $|N(v) \cap W|$.
- The quantity $|N(v) \cap (W \cup B_{high})|$.
- The counter c_v.

At initialization, every vertex v is in W (so D and B are empty). Consequently, the quantities $|N(v) \cap W|$ and $|N(v) \cap (W \cup B_{high})|$ are both equal to $\deg(v)$. For each vertex v, if $\deg(v) \leq \delta\alpha$, then $v \in W_{low}$. Each counter c_v is initialized to 0.

It will be useful to define the graph G_{low}, which changes over the execution of the algorithm:

▶ **Definition 11.** *Let G_{low} be the graph with vertex set W_{low} such that there is an edge (u, v) in G_{low} if there is a path of length 2 between u and v in $G[W \cup B_{high}]$.*

The algorithm proceeds as follows. Repeat the following until W is empty. Compute an MIS I with respect to G_{low}. This step is implemented differently for Theorem 4 and Theorem 10, and we describe the details of this step later.

Then, each vertex in I adds itself to D and tells its neighbors to increment their counters. Whenever the counter of a vertex reaches $\delta\alpha$, it enters D (and does *not* tell its neighbors to increment their counters).

Whenever a vertex moves from one set of the partition to another, it notifies each of its neighbors v so that v can update the quantities $|N(v) \cap W|$ and $|N(v) \cap (W \cup B_{high})|$, and move to the appropriate set. When no more vertices are left in W, B_{high} is also empty, and all processors terminate. This concludes the description of the algorithm. See Algorithm 2 for the precise ways that vertices react to the messages that they receive.

4.1.3 MIS subroutine

Theorem 4 is a reduction from MDS to MIS, while Theorem 10 is not, so we need to describe the MIS subroutine (in the CONGEST model) only for Theorem 10. Recall that we cannot use a reduction to MIS in the CONGEST model because running an MIS algorithm directly on G_{low} could result in messages that become too large after translating the algorithm to run on G.

Our goal is to compute an MIS with respect to G_{low}, using small messages sent over G. We use a simple adaptation of Luby's algorithm. Recall that Luby's algorithm builds an MIS I as follows. While the graph is non-empty, do the following: Add all singletons to I. Then, each vertex v picks a random value $p(v) \in [0, 1]$. Then, all vertices whose value is less than that of all of their neighbors are added to I. Then, all vertices that are in I or have a neighbor in I are removed from the graph for the next iteration of the loop.

We use the following adaptation of Luby's algorithm. See Algorithm 3 for the pseudocode. Initially, the set L of *live* vertices is the set W_{low}. While $L \neq \emptyset$, do the following: Each vertex $v \in L$ picks a random value $p(v) \in [0, 1]$. In the first round each $v \in L$ sends $p(v)$ to its neighbors. In the second round, each vertex that receives one or more values $p(v)$, forwards to its neighbors the minimum value that it received. Then, for each vertex $v \in W_{low}$, if $p(v)$ is equal to the minimum value that v receives in the second round, v is added to I. When v is added to I, v notifies its neighbors, and each neighbor of v that is in $W \cup B_{high}$ forwards this notification to their neighbors. Note that each vertex has at most one neighbor in I, so forwarding this notification only takes one round. Now, every vertex knows whether it has a neighbor with respect to G_{low} that is in I, and every vertex that does is removed from L for the next iteration of the loop.

The proof that this algorithm runs in $O(\log n)$ rounds with high probability and produces an MIS with respect to G_{low} is the same as the analysis of Luby's algorithm and we will not include it here.

4.2 Analysis

The proof that Algorithm 2 achieves an $O(\alpha)$-approximation is precisely the same as that of the centralized algorithm (see Section 3.2.1) given that no counter c_v ever exceeds $\delta\alpha$. This is true because in a single iteration of the **while** loop each vertex can only have its counter incremented once since only vertices in the MIS I send INCREMENT COUNTER messages, and each vertex in $W \cup B_{high}$ only has at most one neighbor in I. This bound on the number of neighbors in I holds, since otherwise there is a path of length 2 between two vertices in $G[W \cup B_{high}]$, making I not an independent set in G_{low}. Once c_v reaches $\delta\alpha$, the vertex v enters D, which prevents c_v from increasing in the future.

Our goal in this section is to prove that if the MIS subroutine takes $R(n)$ rounds, then Algorithm 2 takes $O(R(n) \cdot \alpha^2 \log n)$ rounds. First, we note that the body of the **while** loop besides the MIS subroutine takes a constant number of rounds. Thus, our goal is to show that the number of iterations of the **while** loop is $O(\alpha^2 \log n)$.

Algorithm 2 Distributed $O(\alpha)$-approximation for MDS using MIS.

```
Initialize partition: $D \leftarrow \emptyset$, $B_{high} \leftarrow \emptyset$, $B_{low} \leftarrow \emptyset$, $W \leftarrow V$, $W_{low} \leftarrow \{v \in V : \deg(v) \leq \delta\alpha\}$
Initialize counters: $\forall v \in V : c_v \leftarrow 0$
Initialize degrees: $\forall v \in V : |N(v) \cap W| = \deg(v)$, $|N(v) \cap (W \cup B_{high})| = \deg(v)$
while $W \neq \emptyset$ do
    Find an MIS $I$ with respect to the graph $G_{low}$
    Each vertex $v$ runs the following procedure:
    if $v \in I$ then
        Move $v$ to $D$
        Send INCREMENT COUNTER message to neighbors
        Send MOVED FROM $W$ TO $D$ message to neighbors
    if $v \in W \cup B_{high}$ and $v$ receives INCREMENT COUNTER then
        Increment $c_v$
        if $c_v = \delta\alpha$ then
            if $v \in W$ then
                Send MOVED FROM $W$ TO $D$ message to neighbors
            if $v \in B_{high}$ then
                Send MOVED FROM $B_{high}$ TO $D$ message to neighbors
            Move $v$ to $D$
    // The rest of the algorithm is bookkeeping
    if $v$ receives MOVED FROM $W$ TO $D$ then
        Decrement $|N(v) \cap W|$
        if $v \in B_{high}$ and $|N(v) \cap W| = \delta\alpha$ then
            Move $v$ to $B_{low}$
    if $v$ receives MOVED FROM $W$ TO $D$ or MOVED FROM $B_{high}$ TO $D$ then
        Decrement $|N(v) \cap (W \cup B_{high})|$
        if $v \in W$ and $|N(v) \cap W| \leq \delta\alpha$ then
            Move $v$ to $B_{low}$
            Send MOVED FROM $W$ TO $B_{low}$ message to neighbors
        else if $v \in W$ and $|N(v) \cap W| > \delta\alpha$ then
            Move $v$ to $B_{high}$
            Send MOVED FROM $W$ TO $B_{high}$ message to neighbors
    if $v$ receives MOVED FROM $W$ TO $B_{low}$ or MOVED FROM $W$ TO $B_{high}$ then
        Decrement $|N(v) \cap W|$
        if $v \in B_{high}$ and $|N(v) \cap W| = \delta\alpha$ then
            Move $v$ to $B_{low}$
    if $v$ receives MOVED FROM $W$ TO $B_{low}$ then
        Decrement $|N(v) \cap (W \cup B_{high})|$
        if $v \in W$ and $|N(v) \cap (W \cup B_{high})| = \delta\alpha$ then
            Add $v$ to $W_{low}$
```

Algorithm 3 Distributed MIS with respect to G_{low} in the CONGEST model.

```
L = W_low$
```

```
$L = W_{low}$
while $L \neq \emptyset$ do
    Each vertex $v$ runs the following procedure:
    if $v \in L$ then
        $p(v) \leftarrow$ a value in $[0,1]$ chosen uniformly at random
        Send $p(v)$ message to neighbors
    Send $m_v = \min_{y \in N(v) \cap L} p(y)$ message to neighbors
    if $p(v) = \min_{y \in N(v)} m_y$ then
        Add $v$ to $I$
        Send ADDED message to neighbors
    if $v \in W \cup B_{high}$ and $v$ receives ADDED then
        Send NEIGHBOR ADDED message to neighbors
    if $v$ receives NEIGHBOR ADDED and $v \in L$ then
        Remove $v$ from $L$
```

We begin with a simple claim about the behavior of the partition of vertices over time:

▷ Claim 12.

(1) No vertex can ever enter W from another set.

(2) No vertex can ever move from W_{low} to W_{high}.

Proof. The proof of item 1 is the same as in the proof of Claim 7. For item 2, it is impossible for a vertex to move from W_{low} to W_{high} since for all v the quantity $N(v) \cap (W \cup B_{high})$ that determines membership in W_{low} versus W_{high}, can only decrease over time (in Algorithm 2, this quantity is only decremented). ◁

We begin with the following claim, which when combined with Claim 12, implies that each vertex only spends a limited number of rounds in W_{low}.

▷ Claim 13. For every vertex v that is ever in W_{low}, within $(\delta\alpha)^2$ iterations of the **while** loop after v joins W_{low}, v leaves W.

Proof. First we note that by Claim 12 no vertex can ever move from W_{low} to W_{high}. Thus, if v is in W_{low}, v will remain in W_{low} until v leaves W. Suppose v is in W_{low} at the beginning of an iteration of the **while** loop. Because I is an MIS with respect to G_{low}, if v does not join I during this iteration, then v has a neighbor $y \in W \cup B_{high}$ such that a neighbor z of y joins I. As a result, z immediately joins D and c_y is incremented. Thus, during every iteration that v remains in W_{low}, a vertex in $N(v) \cap (W \cup B_{high})$ has its counter incremented. Recall that whenever a vertex has its counter incremented $\delta\alpha$ times, it joins D. Because $v \in W_{low}$, we have that $|N(v) \cap (W \cup B_{high})| \leq \delta\alpha$. Therefore, the event that a vertex in $N(v) \cap (W \cup B_{high})$ has its counter incremented can only happen at most $(\delta\alpha)^2$ times. Thus, v can only remain in W_{low} for $(\delta\alpha)^2$ iterations of the **while** loop. ◁

We will complete the analysis using the fact that enough vertices are in W_{low} at any given point in time. In particular, Claim 6 implies that at least half of the vertices in $W \cup B_{high}$ are in W_{low}. This implies that at least half of the vertices in W are in W_{low}. Formally, we divide the execution of the algorithm into phases where each phase consists of $(\delta\alpha)^2$ iterations of the **while** loop. At the beginning of any phase, at least half of the vertices in W are in W_{low}. By the end of the phase, all of these vertices have left W by Claim 13. Therefore, each phase witnesses at least half of the vertices in W leaving W. By Claim 12, no vertex can re-enter W, so there can only be $O(\log n)$ phases.

Putting everything together, there are $O(\log n)$ phases, each consisting of $(\delta\alpha)^2$ iterations of the **while** loop, and one iteration of the **while** loop takes $O(R(n))$ rounds. Therefore, the total number of rounds is $O(R(n) \cdot \alpha^2 \log n)$.

For Theorem 10, $R(n) = O(\log n)$, so the number of rounds is $O(\alpha^2 \log^2 n)$.

References

1 Zeyuan Allen-Zhu and Lorenzo Orecchia. Nearly linear-time packing and covering lp solvers. *Mathematical Programming*, 175(1):307–353, 2019.

2 Saeed Akhoondian Amiri. Deterministic congest algorithm for mds on bounded arboricity graphs. *arXiv preprint*, 2021. arXiv:2102.08076.

3 Saeed Akhoondian Amiri, Stefan Schmid, and Sebastian Siebertz. Distributed dominating set approximations beyond planar graphs. *ACM Transactions on Algorithms (TALG)*, 15(3):1–18, 2019.

4 Srinivasa R Arikati, Anil Maheshwari, and Christos D Zaroliagis. Efficient computation of implicit representations of sparse graphs. *Discrete Applied Mathematics*, 78(1-3):1–16, 1997.

5 Brenda S Baker. Approximation algorithms for np-complete problems on planar graphs. *Journal of the ACM (JACM)*, 41(1):153–180, 1994.

6 Nikhil Bansal and Seeun William Umboh. Tight approximation bounds for dominating set on graphs of bounded arboricity. *Information Processing Letters*, 122:21–24, 2017.

7 Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

8 Suman K Bera, Amit Chakrabarti, and Prantar Ghosh. Graph coloring via degeneracy in streaming and other space-conscious models. *arXiv preprint*, 2019. arXiv:1905.00566.

9 Suman K. Bera, Noujan Pashanasangi, and C. Seshadhri. Linear time subgraph counting, graph degeneracy, and the chasm at size six. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 38:1–38:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ITCS.2020.38.

10 Suman K. Bera and C. Seshadhri. How the degeneracy helps for triangle counting in graph streams. In Dan Suciu, Yufei Tao, and Zhewei Wei, editors, *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, pages 457–467. ACM, 2020. doi:10.1145/3375395.3387665.

11 Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 1999. doi:10.1007/3-540-48447-7_34.

12 Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047, 2000.

13 Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.

14 Miroslav Chlebík and Janka Chlebíková. Approximation hardness of dominating set problems in bounded degree graphs. *Information and Computation*, 206(11):1264–1275, 2008.

15 Andrzej Czygrinow, Michał Hańćkowiak, and Edyta Szymańska. Fast distributed approximation algorithm for the maximum matching problem in bounded arboricity graphs. In *International Symposium on Algorithms and Computation*, pages 668–678. Springer, 2009.

16 Andrzej Czygrinow, Michal Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *International Symposium on Distributed Computing*, pages 78–92. Springer, 2008.

17 Janosch Deurer, Fabian Kuhn, and Yannic Maus. Deterministic distributed dominating set approximation in the congest model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 94–103, 2019.

18 Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. A new multilayered PCP and the hardness of hypergraph vertex cover. *SIAM J. Comput.*, 34(5):1129–1146, 2005. doi:10.1137/S0097539704443057.

19 Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 624–633, 2014.

20 Talya Eden, Reut Levi, and Dana Ron. Testing bounded arboricity. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2081–2092. SIAM, 2018. doi:10.1137/1.9781611975031.136.

21 Talya Eden, Dana Ron, and Will Rosenbaum. The arboricity captures the complexity of sampling edges. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 52:1–52:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.52.

22 Talya Eden, Dana Ron, and C. Seshadhri. Faster sublinear approximation of the number of k-cliques in low-arboricity graphs. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1467–1478. SIAM, 2020. doi:10.1137/1.9781611975994.89.

23 David Eppstein. Arboricity and bipartite subgraph listing algorithms. *Information processing letters*, 51(4):207–211, 1994.

24 Guy Even, Mohsen Ghaffari, and Moti Medina. Distributed set cover approximation: Primal-dual with optimal locality. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

25 Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354, 1989.

26 Michael L Fredman and Dan E Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 719–725. IEEE, 1990.

27 Michael R Garey. A guide to the theory of np-completeness. *Computers and intractability*, 1979.

28 Mohsen Ghaffari, Christoph Grunau, and Václav Rozhoň. Improved deterministic network decomposition. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2904–2923. SIAM, 2021.

29 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

30 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 784–797, 2017.

31 Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2505–2523. SIAM, 2017. doi:10.1137/1.9781611974782.166.

32 Gaurav Goel and Jens Gustedt. Bounded arboricity to determine the local structure of sparse graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 159–167. Springer, 2006.

33 Meng He, Ganggui Tang, and Norbert Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In Hee-Kap Ahn and Chan-Su Shin, editors, *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, volume 8889 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 2014. doi:10.1007/978-3-319-13075-0_11.

34 Lujun Jia, Rajmohan Rajaraman, and Torsten Suel. An efficient distributed algorithm for constructing small dominating sets. *Distributed Computing*, 15(4):193–205, 2002.

35 David S Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.

36 Mark Jones, Daniel Lokshtanov, MS Ramanujan, Saket Saurabh, and Ondřej Suchỳ. Parameterized complexity of directed steiner tree on sparse graphs. In *European Symposium on Algorithms*, pages 671–682. Springer, 2013.

37 Haim Kaplan and Shay Solomon. Dynamic representations of sparse distributed networks: A locality-sensitive approach. *ACM Transactions on Parallel Computing (TOPC)*, 8(1):1–26, 2021.

38 David R Karger, Philip N Klein, and Robert E Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328, 1995.

39 Christos Koufogiannakis and Neal E Young. A nearly linear-time ptas for explicit fractional packing and covering linear programs. *Algorithmica*, 70(4):648–674, 2014.

40 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22–26, 2006*, pages 980–989. ACM Press, 2006. doi:10.5555/1109557.1109666.

41 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local computation: Lower and upper bounds. *Journal of the ACM (JACM)*, 63(2):1–44, 2016.

42 Fabian Kuhn and Roger Wattenhofer. Constant-time distributed dominating set approximation. *Distributed Computing*, 17(4):303–310, 2005.

43 Christoph Lenzen and Roger Wattenhofer. Minimum dominating set approximation in graphs of bounded arboricity. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2010. doi:10.1007/978-3-642-15763-9_48.

44 Andrew McGregor and Sofya Vorotnikova. A simple, space-efficient, streaming algorithm for matchings in low arboricity graphs. In Raimund Seidel, editor, *1st Symposium on Simplicity in Algorithms, SOSA 2018, January 7-10, 2018, New Orleans, LA, USA*, volume 61 of *OASICS*, pages 14:1–14:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/OASIcs.SOSA.2018.14.

45 Adir Morgan, Shay Solomon, and Nicole Wein. Algorithms for the minimum dominating set problem in bounded arboricity graphs: Simpler, faster, and combinatorial. *arXiv preprint*, 2021. arXiv:2102.10077.

46 Jose C Nacher and Tatsuya Akutsu. Minimum dominating set-based methods for analyzing biological networks. *Methods*, 102:57–63, 2016.

47 Krzysztof Onak, Baruch Schieber, Shay Solomon, and Nicole Wein. Fully dynamic MIS in uniformly sparse graphs. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 92:1–92:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.92.

48 Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 220–239. SIAM, 2016.

49 David Peleg and Shay Solomon. Dynamic $(1+\epsilon)$-approximate matchings: A density-sensitive approach. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 712–729. SIAM, 2016. `doi:10.1137/1.9781611974331.ch51`.

50 Kent Quanrud. Nearly linear time approximations for mixed packing and covering problems without data structures or randomization. In *Symposium on Simplicity in Algorithms*, pages 69–80. SIAM, 2020.

51 Václav Rozhoň and Mohsen Ghaffari. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 350–363, 2020.

52 Chao Shen and Tao Li. Multi-document summarization via the minimum dominating set. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 984–992, 2010.

53 Shay Solomon and Nicole Wein. Improved dynamic graph coloring. In *26th Annual European Symposium on Algorithms (ESA 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

54 Hsin-Hao Su and Hoa T. Vu. Distributed dense subgraph detection and low outdegree orientation. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.DISC.2020.15`.

55 Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

56 Peng-Jun Wan, Khaled M Alzoubi, and Ophir Frieder. Distributed construction of connected dominating set in wireless ad hoc networks. In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1597–1604. IEEE, 2002.

57 Neal E Young. Nearly linear-work algorithms for mixed packing/covering and facility-location linear programs. *arXiv preprint*, 2014. `arXiv:1407.3015`.

Smoothed Analysis of Population Protocols

Gregory Schwartzman
JAIST, Nomi, Japan

Yuichi Sudo
Hosei University, Tokyo, Japan

Abstract

In this work, we initiate the study of *smoothed analysis* of population protocols. We consider a population protocol model where an adaptive adversary dictates the interactions between agents, but with probability p every such interaction may change into an interaction between two agents chosen uniformly at random. That is, p-fraction of the interactions are random, while $(1-p)$-fraction are adversarial. The aim of our model is to bridge the gap between a uniformly random scheduler (which is too idealistic) and an adversarial scheduler (which is too strict).

We focus on the fundamental problem of leader election in population protocols. We show that, for a population of size n, the leader election problem can be solved in $O(p^{-2}n\log^3 n)$ steps with high probability, using $O((\log^2 n)\cdot(\log(n/p)))$ states per agent, for *all* values of $p \leq 1$. Although our result does not match the best known running time of $O(n\log n)$ for the uniformly random scheduler ($p=1$), we are able to present a *smooth transition* between a running time of $O(n \operatorname{polylog} n)$ for $p=1$ and an infinite running time for the adversarial scheduler ($p=0$), where the problem cannot be solved. The key technical contribution of our work is a novel *phase clock* algorithm for our model. This is a key primitive for much-studied fundamental population protocol algorithms (leader election, majority), and we believe it is of independent interest.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Dynamic graph algorithms

Keywords and phrases Population protocols, Smoothed analysis, Leader election

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.34

Related Version *Full Version*: https://arxiv.org/abs/2105.12262

1 Introduction

In the traditional population protocol model [5], we have a population of n agents, where every agent is a finite state machine with a small number of states. We refer to the cross product of all of the states of the agents in the population as the *configuration* of the population. Two agents can interact, whereupon their internal states may change as a *deterministic* function of their current states. While the transition function is deterministic, it need not be symmetrical. That is, the interaction is ordered, where one agent is called an *initiator* and the other is called a *responder*. A standard assumption is that the order of agents upon an interaction is chosen uniformly at random. This is equivalent to having a single random bit that can be used by the transition function.

The sequence of interactions that the population undergoes is called a *schedule*, and is decided by a *scheduler*. The standard scheduler used in this model is the *uniformly random* scheduler, which chooses all interactions uniformly at random. Agent states are mapped to outputs via a problem specific output function. Generally, a protocol aims to take any legal initial configuration (legal input) and, after a sufficient number of interactions, turn it into one of a desired set of configurations (legal output). After the population reaches a legal output configuration, every following configuration is also a legal output. The running time of the protocol is an upper bound on the number of interactions (steps) required to map any legal input to a legal output. The model assumes agent interactions happen sequentially, but another common term is *parallel time*, which is the running time of a protocol divided by n. A formal definition of population protocols is given in Section 2.

In this paper, we focus on the *leader election* problem. In this problem, every agent is initially marked as either a *leader* or a *follower*. We are guaranteed that initially there exists at least one leader in the population. The goal is to design a protocol, such that, after a sufficient number of interactions, the population always converges to a configuration with a unique leader. The leader election problem has received a large amount of attention in the population protocol literature [5, 4, 3, 1, 2, 26, 27, 38, 37, 39, 19, 20, 40, 42, 25, 7, 13, 14, 35] and thus makes the perfect case study for our model.

Motivation for our model. Population protocols aim to model the computational power of a population of many weak computational entities. Initially introduced to model animal populations [5] (a flock of birds, each attached with a sensor), the model has found use in a wide range of fields. For example: wireless sensor networks [31, 24], molecular computation (e.g. DNA computing) [21, 18, 16]. The assumption of completely uniform interactions in these models is a reasonable *approximation* to the true nature of the interactions. That is, a flock of birds does not interact uniformly at random, the interaction probability of molecules in a fluid can depend on their size and shape, and sensors in a sensor network may experience delays, malfunctions, or even adversarial attacks. The common thread among all of these scenarios is that, while the uniformity assumption is too strong, these systems still contain some amount of randomness. There is a rich literature on designing population protocols for the uniformly random scheduler, and it would be very disheartening if these results do not generalize if we slightly weaken the scheduler. In this paper we try to model these environments, which are "somewhat noisy", and answer the question: Are current population protocol algorithms robust or fragile?

Smoothed analysis. To this end we consider *smoothed analysis* of population protocols. Smoothed analysis was first introduced by Spielman and Teng [34, 33], in an attempt to explain the fact that some problems admit strong theoretical lower bounds, but in practice are solved on a daily basis. The explanation smoothed analysis suggests for this gap, is that lower bounds are proved using very specific, pathological instances of a problem, which are highly unlikely to happen in practice. They support this idea by showing that some lower bound instances are extremely *fragile*, i.e., a small random perturbation turns a hard instance into an easy one. Spielman and Teng applied this idea to the simplex algorithm, and showed that, while requiring an exponential time in the *worst case*, if we apply a small random noise to our instance before executing the simplex algorithm on it, the running time becomes polynomial in expectation.

While in classical algorithm analysis worst-case analysis currently reigns supreme, the opposite is true regarding population protocols. The vast majority of population protocols assume the uniformly random scheduler. This is due to the fact that under the adversarial scheduler most problems of interest are pathological. This reliance on the uniformly random scheduler leads us to ask the following questions: Is the assumption regarding a uniformly random scheduler too strong? Will the algorithms developed under this assumption fail in the real world? We use smoothed analysis to show that indeed it is possible to design *robust* algorithms for the much-studied leader election problem in population protocols. That is, an algorithm that can provide convergence guarantees even if only a *tiny* percentage of the interactions is random. In doing so we smoothly bridge the gap between the adversarial scheduler and the uniformly random scheduler.

Our model and results. It is easy to see that if *all* of the interactions are chosen adversarially, no problem of interest can be solved. In this paper, we present a model which smoothly bridges the gap between the adversarial scheduler and the uniformly random scheduler. In our model, we have an adaptive adversary which chooses the $(i+1)$-th interaction for the population after the completion of the i-th interaction. This choice can be based on *any* past information of the system (interactions, configurations, random bits flipped). With probability $1-p$ the next interaction is the one chosen by the adversary, and with probability p it is an interaction between two agents chosen uniformly at random. We call p the smoothing parameter. In our analysis we allow protocols to access randomization directly as in [15, 29, 11]. Specifically, we assume that each time two agents interact, they get one (unbiased) random bit. In the appendix, we show how to extend our results even if we only assume that the order of initiator-responder is random (and no random bit is given). This results in a slight slowdown in our convergence time by a $O(p^{-1}\log n)$ multiplicative factor. If we assume a random bit is flipped for every interaction, then the adversary cannot decide the outcome of the random bit (but may decide the initiator-responder order). While if we assume no random bit is flipped, then the adversary cannot decide the initiator-responder order, and it is taken to be random. This model is meant to model an environment that is mostly adversarial, but contains a small amount of randomness.

Throughout this work, we assume that agents know some lower bound for the smoothing parameter p (see section 2 for more details). This might seem like a strong assumption at first glance. Let us provide two examples to motivate this assumption.

- Population protocols are often motivated by biological systems, for example, viruses interacting in a fluid. These biological systems undergo an evolutionary process that allows them to learn the value p as they evolve. Imagine several populations of viruses, each with some different estimate of p. The populations that underestimate or overestimate p will die out, while those with a reasonable estimate of p will remain.
- Consider the case of small artificial agents, such as nanobots or sensors mounted on birds. As these agents are deployed to a physical environment, it is possible to measure the environment beforehand and get some estimate of p. If a direct measurement is impossible, a trial and error approach of guessing p might be sufficient (recall that we only need a reasonable lower bound).

We consider the fundamental problem of leader election in this model and show that we can design a protocol for our model which uses $O((\log^2 n)\cdot(\log(n/p)))$ states per agent, and elects a unique leader in $O(p^{-2}n\log^3 n)$ steps with high probability. Although our result does not match the best known running time of $O(n\log n)$, using $O(\log\log n)$ states, for the uniformly random scheduler ($p=1$), we are able to present a *smooth transition* between a running time of $O(n \operatorname{polylog} n)$, using $O(\operatorname{polylog} n)$ states, for $p=1$ and an infinite running time for the adversarial scheduler ($p=0$), where the problem cannot be solved, regardless the number of states.

Furthermore, this shows that *any* amount of noise in the system is sufficient to guarantee that the leader election problem can be solved if we allow for a sufficient number of states per agent. We also note that because the number of states required is $O((\log^2 n)\cdot(\log(n/p)))$, even for an extremely minuscule amount of noise, $p=1/poly(n)$, the leader election problem can be solved by agents using $\operatorname{polylog}(n)$ states. This is important because we would like our agents to be very simple computational units, so we would like to avoid agents with a super-polylogarithmic number of states.

The key building block in our leader election algorithm is the *phase clock* primitive (see section 3 for a formal definition). This is a weak synchronization primitive, which is at the heart of many state of the art algorithms for fundamental problems like leader election and majority in population protocols [6, 2, 26, 27, 11, 38, 35, 9]. The analysis for all current *phase clock* implementations fails for any constant smoothing parameter $p < 1$, assuming an $O(\text{polylog}(n))$ number of states. Roughly speaking, existing phase clocks break when the adversary chooses two agents and repeatedly forces them to interact (a detailed explanation is given in Section 3.1).

We present a novel phase clock design that is robust even when all but a tiny fraction of the interactions are adversarial. Our phase clock relies heavily on the fact that the random bits flipped per interaction are not chosen in an adversarial fashion. To overcome the shortcomings of existing phase clock algorithms, we base our phase clock on a stochastic process whose correctness is *indifferent* to adversarial interactions. Finally, we show that using our phase clock in a simplified (and slower) version of the leader election algorithm of [38] achieves the desired running time. Although we provide a complete (and simplified) proof of the leader election protocol with our phase clock for completeness, the original analysis [38] still goes through unchanged. That is, our phase clock is basically plug-and-play. Thus, we believe this primitive can be used directly for more complex population protocols such as the complete leader election algorithm of [38], or the majority algorithm of [2, 9]. However, properly presenting and analyzing these algorithms is beyond the scope of this paper, and we leave it for future work.

1.1 Related Work

Smoothed analysis was introduced by Spielman and Teng [34, 33]. Since then, it has received much attention in sequential algorithm design (see the survey in [34]). Recently, smoothed analysis has also received some attention in the distributed setting. The first such application is due to Dinitz et al. [22], who apply it to various well-studied problems in dynamic networks. Since then, different smoothing models [28] and problems [17, 30] were considered. To the best of our knowledge, we are the first to consider smoothed analysis of population protocols.

Leader election has been extensively studied in the population protocol model. The problem was first considered in [5], where a simple protocol was presented. In this protocol, all agents are initially leaders, and we have only one transition rule: when two leaders meet, one of them becomes a follower (i.e., a non-leader). This simple protocol uses only two states per agent and elects a unique leader in $O(n^2)$ steps in expectation. This protocol is time-optimal: Doty and Soloveichik [23] showed that any constant space protocol requires $\Omega(n^2)$ expected steps to elect a unique leader. In a breakthrough result, Alistarh and Gelashvili [3] designed a leader election protocol that converges in $O(n \log^3 n)$ expected steps and uses $O(\log^3 n)$ states per agent. Thereafter, a number of papers have been devoted to fast leader election [2, 26, 27, 38, 11]. Gąsieniec, Staehowiak, and Uznanski [27] gave an algorithm that converges in $O(n \log n \log\log n)$ expected steps and uses a surprisingly small number of states: only $O(\log\log n)$ states per agent. This is space-optimal because it is known that every leader election protocol with a $O(n^2/\text{polylog}(n))$ convergence time requires $\Omega(\log\log n)$ states [1]. Sudo et al. [38] gave a protocol that elects a unique leader within $O(n \log n)$ expected steps and uses $O(\log n)$ states per agent. This is time-optimal because any leader election protocol requires $\Omega(n \log n)$ expected steps even if it uses an arbitrarily large number of states and the agents know the exact size of the population [36]. These two protocols were the state-of-the-art until recently, when Berenbrink et al. [11] gave a time and space optimal protocol. In all of the above literature, the stabilization time (i.e., the number of steps it takes to elect a unique leader) is evaluated under the uniformly random scheduler.

Self-stabilizing leader election has also been well studied [4, 37, 39, 19, 20, 40, 42, 25, 7, 13, 14, 35]. In the self-stabilizing setting, we do not assume that all agents are initialized at the beginning of an execution. That is, we must guarantee that a single leader is elected eventually and maintained thereafter even if the population begins an execution from an *arbitrary* configuration. Typically, the population must create a new leader if there is no leader initially, while the population must decrease the number of leaders to one if there are two or more leaders initially. Unfortunately, the self-stabilizing leader election cannot be solved in the standard model [4]. Thus, this problem has been considered (i) by assuming that the agents have global knowledge such as the exact number of agents [14, 13, 40], (ii) by assuming the existence of oracles [25, 7], (iii) by slightly relaxing the requirement of self-stabilization [37, 39, 35], or (iv) by assuming a specific topology of the population such as rings [4, 19, 20, 42].

Several papers on population protocols assume the globally fair scheduler [5, 4, 25, 7, 19]. Intuitively, this scheduler cannot avoid a possible step forever. Formally, the scheduler guarantees that in an infinite execution, a configuration appears infinitely often if it is reachable from a configuration that appears infinitely often. Assuming the fairness condition is very helpful in designing protocols that solve some problem *eventually*, however, it is not helpful in bounding the stabilization time. Thus, the uniformly random scheduler is often assumed to evaluate the time complexities of protocols, as mentioned above. Actually, the uniformly random scheduler is a special case of the globally fair scheduler.

Several papers considered population protocols with some form of noise. In [41] a random scheduler with non-uniform interaction probabilities is proposed, and the problem of data collection is analyzed for this model. While their model generalizes the standard random scheduler, it still does not allow adversarial interactions, and thus is quite different from our model. Sadano et al. [32] introduced and considered a stronger model than the original population protocols under the uniformly random scheduler. In their model, agents can control their moving speeds. Faster agents have a higher probability to be selected by the scheduler at each step. They show that some protocols have a much smaller stabilization time by changing the speeds of agents.

Similarly to us, the authors of [8] also try to answer the question of whether population protocols can function under imperfect randomness. They take a very general approach, which is somewhat different than ours. The main difference is that the randomness of a schedule (a sequence of interactions) is measured as its *Kolmogorov complexity* (the size of the shortest Turing machine which outputs the schedule). Intuitively, the schedule is random if its Kolmogorov complexity is (almost) equal to the length of the schedule. They parameterize the "randomness" of the schedule by a parameter T, where $T = 1$ means that the schedule is completely random, while the randomness decreases as the value of T goes to 0. An *adversary* with parameter T is a scheduler that only generates schedules with parameter T.

They show that any problem which can be solved for $T = 1$ can also be solved for $T < 1$ (imperfect randomness). They also consider the leader election problem, and give upper and lower bounds for the value of T required to solve the problem. Their bounds are not explicit, but are presented as a function of the largest root of a certain polynomial. Apart from a different notion of randomness, their work differs from ours in that it assumes an oblivious adversary (while we consider an adaptive adversary). This makes a direct comparison between our results and those of [8] somewhat tricky. It might be said that we take a somewhat more pragmatic approach, showing a very natural augmentation to the popular random scheduler. This allows us to *explicitly* express the running time and number of states for all values of p (showing that for reasonable values, the performance is very close to that of the random scheduler), while in [8] only existence results are presented.

2 Preliminaries

Population Protocols. A *population* is a network consisting of *agents*. We denote the set of all agents by V and let $n = |V|$. We assume that a population is a complete graph, thus every pair of agents (u, v) can interact, where u serves as the *initiator* and v serves as the *responder* of the interaction. Throughout this paper, we use the phrase "with high probability" to denote a probability of $1 - O(n^{-\alpha})$ for an arbitrarily large constant α.

A *protocol* $P(Q, T, X, Y, \pi_{\text{in}}, \pi_{\text{out}})$ consists of a finite set Q of states, a transition function $T : Q \times Q \times \{0, 1\} \to Q \times Q$, a finite set X of input symbols, a finite set Y of output symbols, an input function $\pi_{\text{in}} : X \to Q$, and an output function $\pi_{\text{out}} : Q \to Y$. The agents are given (possibly different) inputs $x \in X$. The input function π_{in} determines their initial states $\pi_{\text{in}}(x)$. When two agents interact, T determines their next states according to their current states and one bit. The *output* of an agent is determined by π_{out}: the output of an agent in state q is $\pi_{\text{out}}(q)$.

A *configuration* is a mapping $C : V \to Q$ that specifies the states of all the agents. We say that a configuration C changes to C' by the interaction $e = (u, v)$ and a bit b, denoted by $C \overset{(e,b)}{\to} C'$, if $(C'(u), C'(v)) = T(C(u), C(v), b)$ and $C'(w) = C(w)$ for all $w \in V \setminus \{u, v\}$.

Thus, given a configuration C, a sequence of interactions (or ordered pairs of agents) $\{\gamma_i\}_{i=0}^{\infty}$, and a sequence of bits $\{b_i\}_{i=0}^{\infty}$, the *execution* starting from C under $\{\gamma_i\}_{i=0}^{\infty}$ and $\{b_i\}_{i=0}^{\infty}$ is defined as the sequence of configurations $\{C_i\}_{i=0}^{\infty}$ such that $C_i \overset{(\gamma_i,b_i)}{\to} C_{i+1}$. A sequence of interactions is called a *scheule* and will be explained in detailed in the next subsection. We assume that each $b_i \in \{0, 1\}$ is a random variable such that $\Pr[b_i = 1] = 1/2$ and these random bits $b_0, b_1, \ldots$ are independent of each other. That is, upon each interaction the two agents have access to a bit of randomness to decide their new states. In the appendix, we show how to extend our results for the more standard population protocol model where only the order of initiator-responder is random, and no additional randomness is available.

Schedulers. A schedule $\gamma = \{\gamma_i\}_{i=0}^{\infty} = \{(u_i, v_i)\}_{i=0}^{\infty}$ is a sequence of ordered pairs which determines the interactions the population of agents undergoes. Note that although γ is ordered, we use a set notation for simplicity. The schedule is determined by a *scheduler*. In the classical population protocol model, a uniformly random scheduler is used. That is, every pair in γ is chosen uniformly at random. Let us denote this scheduler by Γ_u. One can also consider an *adversarial* scheduler. Such a scheduler creates γ in an adversarial fashion. Let us denote this scheduler by Γ_a. We would like to note that while the sequence of interactions is chosen adversarially by Γ_a, it does not determine the coin flips observed by the agents. This type of scheduler can either be *adaptive* or *oblivious* (non-adaptive). In both cases the adversary has complete knowledge of the initial state of the population and the algorithm executed by the agents. However, for the oblivious case the sequence of interactions, γ, must be chosen *before* the execution of the protocol, while for the adaptive case the interaction γ_i is chosen by the adversary after the execution of the $(i - 1)$-th step, with full knowledge of the current state of the population. The difference between an oblivious and an adaptive adversary can also be stated in term of knowledge of the randomness in the population. An adaptive adversary has full knowledge regarding the population, including the random coins used in the past. While the oblivious adversary does not have access to the randomness of the system.

Our model. We consider a smoothed scheduler Γ_s which is a combination of Γ_u and Γ_a. Specifically, let $\gamma^a = \{\gamma_i^a\}_{i=0}^{\infty}, \gamma^u = \{\gamma_i^u\}_{i=0}^{\infty}$ be the schedules chosen by Γ_a, Γ_u. We define the smoothed schedule $\gamma_s = \{\gamma_i^s\}_{i=0}^{\infty}$ of Γ_s as $\gamma_i^s = \gamma_i^u$ with probability p and $\gamma_i^s = \gamma_i^a$ with probability $1-p$, where $p \in [0,1]$ is the *smoothing parameter*. We note that if Γ_a is adaptive, then so is Γ_s. For the rest of the paper we focus on an *adaptive* adversary.

In this paper, we assume that a rough knowledge of an upper bound of n, p^{-1} is available. Specifically, we assume that all agents know two common values n', p', such that $n \leq n' = O(n), p \geq p' = \Omega(p)$. Assuming such a rough knowledge about n is standard in the recent population protocol literature [3, 1, 2, 27, 29, 12, 11, 37, 38, 39, 35, 9], and we generalize this assumption for p. Due to the asymptotic equivalence between n, p and n', p', we only use only n, p in the definition and analysis of our algorithm.

Leader election. The leader election problem requires that every agent should output L or F ("leader" or "follower") respectively. We say that a configuration C of P is output-stable if no agent may change its output in an execution of P that starts from C, regardless of the choice of interactions. Let $\mathcal{S}_P$ be the set of the output-stable configurations such that, for any configuration $C \in \mathcal{S}_P$, exactly one agent outputs L (i.e., is a leader) in C. This problem does not require inputs for the agents. Hence, we assume $X = \{x\}$, thus all the agents begin an execution with a common state $s_{\text{init}} = \pi_{\text{in}}(x)$. We say that a protocol P is a leader election protocol for a scheduler Γ, if the execution of the protocol starting from the configuration where all agents are in state s_{init} reaches a configuration in $\mathcal{S}_P$ with probability 1 with respect to the scheduler Γ. We define the stabilization time of the execution as the number of steps until it reaches a configuration in $\mathcal{S}_P$ for the first time.

One-way Epidemic. In the proposed protocol, we often use the *one-way epidemic* protocol[6]. This is a population protocol where every agent has two states $\{0,1\}$ and the transition function is given as $(x,y) \rightarrow (x, \max\{x,y\})$. All nodes with value 1 are *infected*, while all nodes with value 0 are *susceptible*. Initially, we assume that a single node is infected. We say that the one-way epidemic finishes when all nodes are infected. This is an important primitive for spreading a piece of information among the population.

Angluin et al. [6] prove that one-way epidemic finishes within $\Theta(n \log n)$ interactions with high probability for Γ_u. It is easy to see that the one-way epidemic protocol finishes within $O(p^{-1} n \log n)$ steps for Γ_s with high probability. This is because within $O(p^{-1} n \log n)$ steps of Γ_s there must exist $\Omega(n \log n)$ *random* interactions with high probability. Due to the nature of the one-way epidemic protocol, we can just ignore all adversarial interactions, and the original analysis goes through.

Martingale concentration bounds. In our analysis we often encounter the following scenario: We have a series of *dependent* binary random variables $\{X_i\}_{i \geq 0}$ such that $\forall i, E[X_i \mid X_0, ..., X_{i-1}] \geq q$, for some constant q.[1] And we would like to bound the probability $Pr[\sum_{i=0}^{\lceil \alpha q^{-1} t \rceil} X_i \leq t]$ for some constant α. Note that if the variables were independent, we could have simply used a Chernoff type bound. As this is not the case, we use martingales for our analysis.

[1] Note that this condition is equivalent to $\forall i, Pr[X_i = 1 \mid X_0 = s_0, ..., X_{i-1} = s_{i-1}] \geq q$ for any binary string s of length $i-1$.

We say that a sequence of random variables, $\{Y_i\}_{i=0}$, is a sub-martingale with respect to another sequence of random variables $\{X_i\}_{i\geq 0}$ if it holds that $\forall i, E[Y_i \mid X_0, ..., X_{i-1}] \geq Y_{i-1}$. The following concentration equality holds for sub-martingales:

▶ **Theorem 1** (Azuma). *Suppose that $\{Y_i\}_{i\geq 0}$ is a sub-martingale with respect to $\{X_i\}_{i\geq 0}$, and that $|Y_i - Y_{i-1}| \leq c_i$. Then for all positive integers k and positive reals ϵ it holds that:*

$$Pr[Y_k - Y_0 < -\epsilon] \leq e^{\frac{-\epsilon^2}{2\sum_{j=1}^{k} c_j}}$$

Let us consider the sequence $\{X_i\}_{i\geq 0}$ from before. Recall that $\forall i, E[X_i \mid X_0, ..., X_{i-1}] \geq q$. Without loss of generality assume that $X_0 = 0$.

Let us define $Y_i = \sum_{j=0}^{i} X_j - q \cdot i$. Note that $Y_0 = 0$. Let us show that $\{Y_i\}_{i\geq 0}$ is a a submartingale with respect to $\{X_i\}_{i\geq 0}$. It holds that:

$$E[Y_{i+1} \mid X_0, ..., X_i] = E[X_{i+1} - q + Y_i \mid X_0, ..., X_i] = E[X_{i+1} \mid X_0, ..., X_i] - q + Y_i > Y_i$$

Where in the transitions we used the fact that the variables $X_0, .., X_i$ completely determine Y_i, thus $E[Y_i \mid X_0, ..., X_i] = Y_i$, and the fact that $E[X_i \mid X_0, ..., X_{i-1}] \geq q$. Next we apply Azuma's inequality for $Y_{\lceil 2q^{-1}t \rceil}$ by setting $\epsilon = t$, noting that $\forall i, |Y_{i+1} - Y_i| \leq 1$.

$$\Pr\left[\sum_{i=0}^{\lceil 2q^{-1}t \rceil} X_i < t\right] \leq \Pr\left[\sum_{i=0}^{\lceil 2q^{-1}t \rceil} X_i - 2t \leq -t\right]$$
$$\leq \Pr[Y_{\lceil 2q^{-1}t \rceil} < -t] \leq e^{-t^2/\lceil 2q^{-1}t \rceil} = e^{-\Theta(t)}$$

We state the following theorem:

▶ **Theorem 2.** *Let $\{X_i\}_{i\geq 0}$ be a series of binary random variables such that $\forall i, E[X_i \mid X_0, ..., X_{i-1}] \geq q$ for some constant q. Then it holds that for every positive integer t: $\Pr\left[\sum_{i=0}^{\lceil 2q^{-1}t \rceil} X_i < t\right] \leq e^{-\Theta(t)}$.*

Specifically, when we set $t = \Theta(\log n)$ with a sufficiently large constant we get a high probability bound.

3 Phase clock implementation for Γ_s

A phase clock is a weak synchronization primitive used in population protocols. In a phase clock we would like all of the agents to have a variable, let's call it *hour*, with the following properties:

1. All agents simultaneously spend $\Omega(f(n))$ steps in the same hour.
2. For every agent an hour lasts $O(g(n))$,

Where the above holds with high probability for every value of hour. Ideally, we desire $f(n) = g(n)$.

We borrow some notation from [10], and define the above more formally. A *round* is a period of time during which all agents have the same *hour* value. Denote by $R_s(i), R_e(i)$ the start and end of round i. Formally, $R_s(i)$ is the interaction at which the last agent reaches hour i, while $R_e(i)$ is the interaction during which the first agent reaches hour $i+1$. We define the length of round i as $L(i) = \max\{0, R_e(i) - R_s(i)\}$. Note that it may be the case that $R_e(i) \leq R_s(i)$, thus a max is needed in the definition. Finally, during these $L(i)$ interactions, all agents have *hour* $= i$. Next we define the stretch of round i as $S(i) = R_e(i) - R_e(i-1)$.

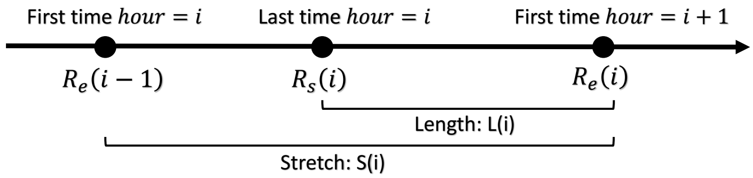


Figure 1 A visual representation of the length and stretch of a round.

This is the amount of time since $hour = i$ is reached for the first time until $hour = i + 1$ is reached for the first time. Note that $L(i) \leq S(i)$ always holds. For a visual representation, we refer the reader to Figure 1. Using the above notation, we define a phase clock.

▶ **Definition 3.** *We say that an algorithm is a phase clock with parameters $f(n)$ and $g(n)$ (or a $(f(n), g(n))$-phase clock) if it has the following guarantees with high probability for any $i \geq 0$:*

1. $L(i) \geq d_1 f(n)$
2. $S(i) \leq d_2 g(n)$

Where d_1 and d_2 are adjustable constants (taken to be sufficiently large). When $f(n) = g(n)$ we simply write a $f(n)$-phase clock.

We note that all current phase clock algorithms require the uniformly random scheduler, and do not extend to Γ_s, as we will see in the next subsection.

3.1 Why existing algorithms fail

In this subsection, we show why the existing phase clock algorithms fail in our model.

There are three kinds of phase clock algorithms in the field of population protocols: a phase clock with a unique leader [6], a phase clock with a junta [26, 27, 11, 9], and a leaderless phase clock [2, 38]. The first kind is essentially a special case of the second kind. The second kind is a $\log n$-phase clock that uses only a constant number of states. However, we require the assumption that there is a set $J \subset V$ of agents marked as members of a *junta*, such that $|J| = O(n^{1-\epsilon})$, where ϵ is a constant. The third kind is a $\log n$-phase clock that uses $O(\log n)$ states but does not require the existence of a junta.

In our notation, the second algorithm can be written as follows:

- Each agent has a variable $minute \in \mathbb{N}$.
- Each agent outputs $hour = \lfloor minute/M \rfloor$, where M is a (sufficiently large) constant.
- Suppose that an initiator u and a responder v interact. The initiator u sets its $minute$ to $\max(u.minute, v.minute + 1)$ if u is in the junta; otherwise to $\max(u.minute, v.minute)$.

By the definition of the algorithm, only an agent in the junta can increase $\max_{v \in V} v.minute$. This fact and the sublinear size of the junta guarantees that the length of each round is $\Omega(n \log n)$ with high probability. However, this guarantee depends on the uniformly random nature of the scheduler. In our model, every interaction is chosen adversarially with probability $1 - p$. The adversary can force two agents in the junta, say u and v, to interact so frequently that every round finishes within $O(1/(1-p))$ steps. Thus, unless $p = 1 - O(1/n)$, i.e., unless the adversary can only choose an extremely small fraction of the interactions, the adversary can always force each round to finish in $o(n)$ steps, i.e., $o(1)$ parallel time. In particular, if $p = 1 - \Omega(1)$, the adversary can always force each round to finish in a constant number of steps.

The third algorithm (the leaderless phase clock) can be written as follows[2]:

- Each agent has variables *hour* $\in \mathbb{N}$ and *minute* $\in \{0, 1, \dots, M\}$, where $M = \Theta(\log n)$ with a sufficiently large hidden constant.
- Suppose that an initiator u and a responder v interact. The initiator u updates its *hour* and *minute* as follows:

$$(u.\mathit{hour}, u.\mathit{minute}) \leftarrow \begin{cases} (v.\mathit{hour}, 0) & \text{if } u.\mathit{hour} < v.\mathit{hour} \\ (u.\mathit{hour}+1, 0) & \text{else if } u.\mathit{minute} = M \\ (u.\mathit{hour}, u.\mathit{minute}+1) & \text{otherwise.} \end{cases}$$

In this phase clock, an agent resets its *minute* to zero each time it increases its *hour*. Once an agent resets its *minute* to zero, it must have no less than M interactions, or interact with an agent whose *hour* is larger than its *hour*, before it increases its *hour*. Thus, one can easily observe that the length of each round is $\Omega(n \log n)$ under the uniformly random scheduler. However, in our model, this does not hold. The adversary can pick two agents and force them to interact frequently, so that every round finishes within $O((\log n)/(1-p))$ steps, i.e., $O((\log n)/(n(1-p)))$ parallel time.

3.2 Our algorithm

We present a $((np^{-1}\log^2 n), (np^{-2}\log^2 n))$-phase clock using $O((\log n) \cdot \log(n/p))$ states per agent, where p is the smoothing parameter for Γ_s.

In our algorithm each agent v has three states: *second*, *minute*, *hour*. Where *hour* is the output variable. The domain of the variables is: *second* $\in \{0, ..., S\}$, *minute* $\in \{0, ..., M\}$ where $S = \log(n/p) + \log\log n + c, c = O(1)$ and $M = \Theta(\log n)$. For simplicity of notation and without loss of generality, we assume that $1/p, \log n, M, S, c$ are all integers. While the domain of *hour* is unbounded in our algorithm, it can easily be taken to be bounded (By using a simple modulo operation [10], or by stopping the counter once it reaches some upper limit [26, 27]). All variables are initialized to 0. For each interaction, we apply Algorithm 1, where u is the initiator and v is the responder. Roughly speaking, the *second* variable follows the following random walk pattern:

$$\mathit{second} \leftarrow \begin{cases} \mathit{second}+1, & \text{with probability } 1/2 \\ 0, & \text{with probability } 1/2 \end{cases}$$

When it reaches S, the *minute* variable is incremented, and *second* is reset back to 0. When *minute* reaches M, the *hour* variable is incremented and both other variables are reset to 0. Finally, the *hour* and *minute* variables are spread via the one-way epidemic process. By doing so, every agent learns the maximum *hour* value in the system, and the maximum *minute* value for its current *hour*.

The main innovation in our algorithm is the increment pattern that the *second* variable undergoes. Our increment pattern guarantees that the *second* variable is robust to adversarial interactions. What dictates the speed of the increment is the *total number* of interactions in the system.

In the following section, we show that indeed our algorithm is a phase clock with round length $\Theta(np^{-1}\log^2 n)$ and stretch of $\Theta(np^{-2}\log^2 n)$.

[2] The implementation of this phase clock slightly differs between [2] and [38]. Here we describe the implementation presented in [38].

Algorithm 1 Phase clock.

```
$M \leftarrow \Theta(\log n), S \leftarrow \log(n/p) + \log\log n + O(1)$
$\forall v \in V, v.second \leftarrow 0, v.minute \leftarrow 0, v.hour \leftarrow 0$
foreach interaction $(u, v)$ do
    $u$ makes a fair coin flip
    if Heads then $u.second \leftarrow u.second + 1$
    else $u.second \leftarrow 0$
    if $u.second = S$ then
        $u.minute \leftarrow u.minute + 1$
        $u.second \leftarrow 0$
    if $u.minute = M$ then
        $u.hour \leftarrow u.hour + 1$
        $u.minute \leftarrow 0$
    //One-way epidemic
    if $u.hour < v.hour$ then
        $u.hour \leftarrow v.hour$
        $u.minute \leftarrow 0$
        $u.second \leftarrow 0$
    if $u.hour = v.hour$ and $u.minute < v.minute$ then
        $u.minute \leftarrow v.minute$
        $u.second \leftarrow 0$
```

3.3 Analysis

Lower bounding $L(i)$. Our first goal is to show that $L(i) \geq \Omega(np^{-1}\log^2 n)$. In order to achieve this, it enough to show that $S(i) \geq \Omega(np^{-1}\log^2 n)$. This is due to the fact at as soon as a new maximum value for *hour* appears in the population, it is spread to all agents via the one-way epidemic process within $O(np^{-1}\log n)$ steps. Let us formalize this claim. Assume that $S(i) = R_e(i) - R_e(i-1) \geq \Omega(np^{-1}\log^2 n)$. On the other hand, due to the one-way epidemic it holds that $R_s(i) - R_e(i-1) \leq O(p^{-1}n\log n)$. Combining these two facts we get that:

$$\begin{aligned} L(i) = R_e(i) - R_s(i) &\geq \Omega(np^{-1}\log^2 n) + R_e(i-1) - R_s(i) \\ &\geq \Omega(np^{-1}\log^2 n) - O(p^{-1}n\log n) = \Omega(np^{-1}\log^2 n) \end{aligned}$$

Thus, for the rest of this section we focus on lower bounding $S(i)$.

As we aim to bound $S(i) = R_e(i) - R_e(i-1)$, for every i, let us assume for the rest of the analysis that $R_e(i-1) = 0$. That is we assume that time 0 is when $v.hour = i$ holds for some agent for the first time. Let $m' = \max_{v \in V, v.hour = i} v.minute$ and let T_k be a random variable such that $m' = k$ holds in the T_k-th step for the first time. Note that $T_0 = 0$ holds with probability 1. We prove the following lemma:

▶ **Lemma 4.** *For $c > 2$, it holds that $Pr(T_{k+1} - T_k > cnp^{-1}\log n) > 1/2$ for any $k = 0, 1, \ldots, M-1$.*

Proof. For m' to increase by 1 starting from time T_k, at least one agent must observe S consecutive heads in its coin flips. Let us consider $cnp^{-1}\log n$ consecutive interactions starting from time 0. Let us denote by x_v the amount of interactions agent v took part in during this time as initiator. Note that $\sum_{v \in V} x_v = cnp^{-1}\log n$. Let us upper bound the probability of agent v seeing S consecutive heads during this time. The probability that agent v sees a sequence of S heads, starting exactly upon its j-th interaction as initiator, and ending upon its $(\min\{j+S, x_v\})$-th interaction as initiator, is upper bounded by 2^{-S}. Note that if $x_v - j < S$ the probability is 0, but the upper bound still holds. Now let us use a union bound over all values of j. This leads to an upper bound of $x_v \cdot 2^{-S}$ for the probability that agent i sees at least S consecutive heads. Recall that $S = \log(n/p) + \log\log n + c$. To finish the proof we apply a union bound over all agents to get an upper bound of

$$\sum_{v \in V} x_v \cdot 2^{-S} = \frac{cnp^{-1}\log n}{2^c np^{-1}\log n} = c \cdot 2^{-c}$$

for the probability of at least one agent seeing S consecutive heads over a period of $cnp^{-1}\log n$ interactions. Finally $Pr(T_{k+1} - T_k > cnp^{-1}\log n) > 1 - c \cdot 2^{-c}$. By setting $c > 2$ we complete the proof. ◀

The above shows that with constant probability the maximum value of *minute* among all agents does not increase too fast. This holds regardless of the value of *hour*. Next, we show that *with high probability*, for every value of $i \geq 0$, the stretch of round i is sufficiently large.

▶ **Lemma 5.** *For every $i \geq 0$, it holds with high probability that $L(i) = \Omega(np^{-1}\log^2 n)$.*

Proof. Fix some $hour = i$, and let X_k be the indicator variable for the event that $T_{k+1} - T_k > cnp^{-1}\log n$. According to Lemma 4, $E[X_k \mid X_0, ..., X_{k-1}] > 1/2$ holds for any $k = 0, 1, ..., M-1$. Let $X = \sum_{k=0}^{M-1} X_k$, and note that $S(i) \geq X \cdot cnp^{-1}\log n$. Ideally we would like to use a Chernoff bound to lower bound X, but unfortunately the $\{X_k\}$ variables are not independent. Thus, we apply Theorem 2 with parameters $q = 1/2, t = M/4$ and get that: $\Pr\left[\sum_{i=0}^{M} X_i < M/4\right] \leq e^{-\Theta(M)}$.

Recall that $M = \Theta(\log n)$, thus by setting M sufficiently large we get that with high probability $S(i) = \Omega(np^{-1}\log^2 n)$. As noted before, this implies that $L(i) = \Omega(np^{-1}\log^2 n)$, which completes the proof. ◀

We note that the lower bound holds even when all interactions are chosen adversarially ($p = 0$). The only reason p appears in the lower bound is due to the definition of S. We continue to prove our upper bound, for which the existence of random interactions is crucial. Specifically, we require the existence of random interactions in order to utilize one-way epidemics.

Upper bounding $S(i)$. In what follows we upper bound $S(i)$ directly. As before, we first consider m', the maximum value of the *minute* variable, and show that it increases sufficiently fast.

▶ **Lemma 6.** *From any configuration where $m' = j < M$ holds, m' increases to $j+1$ within $dnp^{-2}\log n$ steps with a constant probability, for a sufficiently large constant d.*

Proof. Without loss of generality, we assume that every agent satisfies $hour = i$ and $minute = j$ in the configuration because the one way epidemic propagates $\max_{v \in V} v.hour$ and $\max_{v \in V, v.hour = i} v.minute$ to all agents within $O(p^{-1}n\log n)$ steps with high probability.

In our algorithm, we can say that each agent $v \in V$ plays a *lottery game* repeatedly. Agent v starts one round of the game each time it sees a tail. If v sees S consecutive heads before the next tail, v *wins* the game in that round. Otherwise, (i.e., if v sees less than S heads before it sees the next tail), v *loses* the game in that round. When an agent sees the next tail (or wins the round), the next round of the game begins. At each round of the game, v wins the game with probability $2^{-S} = \Omega(p/(n \log n))$ (Recall that $S = \log(np^{-1}) + \log\log n + O(1)$). The goal of our proof is to show that with constant probability, some agent wins the game at least once within $dnp^{-2} \log n$ steps, for a sufficiently large constant d.

For an agent v, we denote by $W_i(v)$ the event that v wins it's i-th game. Note that the $W_i(v)$ events are independent of each other for all values of i and v. This is because for every interaction only the initiator flips a coin, and the coins used for every game don't overlap. Let us also denote by $Y_k(v) = \bigvee_{i=1}^{k} W_i(v)$, the event that agent v wins at least once in its first k games. Let $Y_k = \bigvee_{v \in V} Y_k(v)$, be the event that at least one agent wins at least one of its first k games.

Let d be a sufficiently large constant and let $\tau = (dp^{-1} \log n)/4$. Let us denote by $X(v)$, the event that agent v plays at least τ games in the first $dp^{-2} n \log n = 4np^{-1}\tau$ steps, and let $X = \bigwedge_{v \in V} X(v)$. Finally we are interested in lower bounding the probability of Z, the event that at least one agent wins a game within the first $4np^{-1}\tau$ steps. We note that it holds that $\Pr[Z] \geq \Pr[X \wedge Y_\tau]$. That is, if all agents play at least τ games within the first $4np^{-1}\tau$ steps, and at least one agents wins one of its first τ games then event Z occurs. Applying a union bound, we write $\Pr[Z] = 1 - \Pr[\neg X \vee \neg Y_\tau] \geq 1 - \Pr[\neg X] - \Pr[\neg Y_\tau]$. In order to conclude the proof we wish to show that $\Pr[\neg Y_\tau] < 1/3$ and $\Pr[\neg X] < 1/3$.

To bound $\Pr[\neg X]$ it is sufficient to show that within the first $4np^{-1}\tau$ steps every agent sees at least τ tails with high probability. As every interaction is chosen uniformly at random with probability p, and the initiator / responder order is also random, within the first $4np^{-1}\tau$ steps each agent will observe at least 2τ tails in expectation. Applying a Chernoff bound for a sufficiently large constant d, we get that every agent will observe τ or more tails w.h.p. Thus $\Pr[\neg X] = O(1/n) < 1/3$.

Next, we bound $\Pr[\neg Y_\tau]$. Expanding the expression, we get:

$$\begin{aligned}\Pr[\neg Y_\tau] &= Pr\left[\bigwedge_{v \in V} \neg Y_\tau(v)\right] = Pr\left[\bigwedge_{v \in V} \bigwedge_{i=1}^{\tau} \neg W_i(v)\right] \\ &= (1 - 2^{-S})^{n\tau} \leq e^{-\frac{n\tau}{2^S}} = e^{-\Omega(d)} \leq 1/3,\end{aligned}$$

where in the above we use the independence of the $\{W_i(v)\}$ events, and the fact that d is sufficiently large. This completes the proof. ◀

We are now ready to prove our upper bound.

▶ **Lemma 7.** *For every $i \geq 0$, it holds with high probability that $S(i) = O(np^{-2} \log^2 n)$.*

Proof. Let d be a sufficiently large constant that satisfies the conditions of Lemma 6 and $\tau = dnp^{-2} \log n$. Fix some *hour* $= i$, and let X_j be the indicator variable such that $X_j = 1$ holds if and only if m' increases at least by one from the $(j-1)\tau$ step to the $\tau j - 1$ step. By Lemma 6, it holds that $E[X_i \mid X_0, ..., X_{i-1}] \geq q$ for some constant q. We finish the proof by applying Theorem 2 with parameter $t = M$ and get that: $\Pr\left[\sum_{i=0}^{2q^{-1}M} X_i < M\right] \leq e^{-\Theta(M)}$.

Recall that $M = \Theta(\log n)$, thus setting M sufficiently large, implies that $S(i) \leq 2q^{-1}M \cdot dnp^{-2} \log n = O(np^{-2} \log^2 n)$ with high probability. ◀

Finally, we state our main theorem:

▶ **Theorem 8.** *Algorithm 1 is a $((np^{-1} \log^2 n), (np^{-2} \log^2 n))$-phase clock that uses $O((\log n) \cdot (\log(p^{-1}n)))$ states for Γ_s.*

4 Leader election

We analyze the following leader election protocol, as described in [38] (the module backup)[3]. In the protocol every agent is either a *a leader* or a *follower*. This is represented via a binary *leader* variable. We assume that initially there is at least one leader in the population. Every agent also has a *level* variable initiated to 0 and bounded by the value $\ell_{max} = \Omega(\log n)$. The protocol assumes the existence of a phase clock in the system. For every agent we call the time between two consecutive increases of the *hour* variable of the phase clock an *epoch for that agent* (this is a subjective value per agent, not to be confused with a *round* as was defined in the previous section). For our usage we can bound the range of the *hour* variable by a small constant. This can be easily implemented via a modulo operation (see [38] for a detailed implementation), where the duration of each round still has the same guarantees of Theorem 8. To simplify the pseudo-code we introduce a *tick* variable which is raised for an agent only in the first interaction it takes part in as an initiator once it enters a new epoch.

The algorithm consists of two parts, where the first part guarantees that we quickly converge to a single leader with high probability, while the second part guarantees that the population *always* reaches a state where there exists a single leader. Accordingly, the second part is very slow to converge, but is rarely required.

1. On the first interaction in each epoch, a leader makes a coin flip and increments the *level* variable if it observed heads (up to the limit ℓ_{max}). Thereafter, the maximum level in the population is shared among all the agents via one-way epidemic. A leader becomes a follower when it observes a higher level than it's own.
2. When two leaders interact, one remains a leader and the other one becomes a follower.

The pseudocode for the above is given in Algorithm 2.

Algorithm 2 Leader election.

```
$\ell_{max} \leftarrow \Theta(\log n)$
$\forall v \in V, v.level \leftarrow 0$
foreach $interaction\ (u, v)$ do
    //One way epidemic
    if $u.level < v.level$ then
        $u.leader \leftarrow false$
        $u.level \leftarrow v.level$
    //$u.tick \leftarrow true$ when $u$ enters a new epoch
    if $u.tick = true$ and $u.leader = true$ then
        $u.tick \leftarrow false$
        $u$ makes a fair coin flip
        if $Heads$ then  $u.level \leftarrow \min\{u.level + 1, \ell_{max}\}$

    if $v.leader = true$ and $u.leader = true$ then $u.leader \leftarrow false$
```

In [38] the correctness and running time are analyzed under Γ_u. First let us present the correctness analysis. That is, there is always at least one leader in the population. Roughly speaking, this is because the first part always keeps the leader with the highest level, while the second part only eliminates a leader if it interacts with another leader. So at any point in time when a leader is eliminated, it can "blame" a leader which currently exists.

[3] Essentially the same idea as [38] was previously presented in [2], however they differ in implementation. We follow the implementation of [38].

As the run-time analysis of [38] is for Γ_u it is not immediately clear what are the implications for Γ_s. Luckily, the analysis still goes through as long as we have a phase clock for Γ_s. Let us present a simplified analysis, which is somewhat different than the analysis presented in [38]. We aim to bound the time it takes to reduce the number of leaders to 1. First we note that because the length of a round is $\Omega(p^{-1}n\log^2 n)$ steps, then with high probability every leader has at least one interaction during the first half of the round, also the information of the interaction is guaranteed to spread to the entire population within the round with high probability. This guarantees that for every round, every leader flips a coin, and the maximum level is propagated throughout the population via the one-way epidemic within that round. For the rest of the analysis we assume that indeed every leader has at least one interaction per epoch and that the phase clock and one-way epidemic function correctly. As these events happen with high probability, we can guarantee that they hold throughout the execution of the first $\Theta(np^{-2}\log^3 n)$ steps with high probability via a simple union bound.[4]

Let us denote by L_i the random variable for the number of leaders remaining after round i, where $L_0 > 0$ is the initial number of leaders. Then it holds that $E[L_i] \leq L_{i-1}(1/2+2^{-L_{i-1}})$. This is because the distribution of L_i behaves exactly like $B(L_{i-1}, 1/2)$ (number of heads when tossing L_{i-1} fair coins), with the exception that if all coins are tails, we get L_{i-1} leaders remaining instead of 0. Thus, when computing the expectation we must add a $L_{i-1}2^{-L_{i-1}}$ term. Finally, note that $L_{i-1}(1/2+2^{-L_{i-1}}) \leq \frac{3}{4}L_{i-1}$ for all $L_{i-1} \geq 2$. Using Markov's inequality, it holds that $Pr[L_i \geq \frac{33}{40}L_{i-1}] = Pr[L_i \geq \frac{3}{4}L_{i-1} \cdot \frac{11}{10}] \leq \frac{10}{11}$.

Let us denote by X_i the indicator random variable for the event that $L_i < \frac{33}{40}L_{i-1}$. Then it holds that $E[X_i \mid X_1, ..., X_{i-1}] > q$ for some constant q. To complete the proof we apply Theorem 2 and get that within $O(\log n)$ epochs we remain with a single leader with high probability. As every epoch requires $O(p^{-2}n\log^2 n)$ steps, we get a unique leader with high probability in $O(p^{-2}n\log^3 n)$ steps. Combining this with the cost of executing the slower second phase of the algorithm up to convergence, we get an expected running time of $O(p^{-2}n\log^3 n)$. This is because the second part requires $O(p^{-1}n^2)$ steps to completes, but is only required if the first part fails, which happens with probability $O(n^{-2})$. Thus, the second part's contribution to the expected stabilization time is $O(1/p)$. We state the following theorem:

▶ **Theorem 9.** *For every $p < 1$, leader election can be solved under Γ_s in $O(p^{-2}n\log^3 n)$ steps with high probability and in expectation using $\Theta((\log^2 n)\cdot(\log(np^{-1})))$ states.*

References

1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2560–2579. SIAM, 2017.

2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.

3 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491, 2015.

[4] Note that when the execution time goes to infinity, these guarantees (i.e., synchronization via a phase clock) eventually fail. But our protocol has long since converged by this time.

4 D. Angluin, J. Aspnes, M. J Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):13, 2008.

5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

6 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

7 J. Beauquier, P. Blanchard, and J. Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *International Conference on Principles of Distributed Systems*, pages 38–52, 2013.

8 Joffroy Beauquier, Peva Blanchard, Janna Burman, and Rachid Guerraoui. The benefits of entropy in population protocols. In *OPODIS*, volume 46 of *LIPIcs*, pages 21:1–21:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

9 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Computing*, pages 1–21, 2020.

10 Petra Berenbrink, Robert Elsässer, Tom Friedetzky, Dominik Kaaser, Peter Kling, and Tomasz Radzik. Time-space trade-offs in population protocols for the majority problem. *Distributed Computing*, pages 1–21, 2020.

11 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 119–129, 2020.

12 Andreas Bilke, Colin Cooper, Robert Elsässer, and Tomasz Radzik. Brief announcement: Population protocols for leader election and exact majority with $O(\log^2 n)$ states and $O(\log^2 n)$ convergence time. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*, pages 451–453, 2017.

13 Janna Burman, David Doty, Thomas Nowak, Eric E Severson, and Chuan Xu. Efficient self-stabilizing leader election in population protocols. *arXiv preprint*, 2019. arXiv:1907.06068.

14 S. Cai, T. Izumi, and K. Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.

15 D. Canepa and M. G. Potop-Butucaru. Stabilizing leader election in population protocols, 2007. URL: http://hal.inria.fr/inria-00166632.

16 Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific reports*, 2(1):1–9, 2012.

17 Soumyottam Chatterjee, Gopal Pandurangan, and Nguyen Dinh Pham. Distributed MST: A smoothed analysis. In *ICDCN*, pages 15:1–15:10. ACM, 2020.

18 Ho-Lin Chen, Rachel Cummings, David Doty, and David Soloveichik. Speed faults in computation by chemical reaction networks. *Distributed Comput.*, 30(5):373–390, 2017.

19 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 53–59, 2019.

20 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election in regular graphs. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 210–217, 2020.

21 Yuan-Jyue Chen, Neil Dalchau, Niranjan Srinivas, Andrew Phillips, Luca Cardelli, David Soloveichik, and Georg Seelig. Programmable chemical controllers made from dna. *Nature nanotechnology*, 8(10):755–762, 2013.

22 Michael Dinitz, Jeremy T Fineman, Seth Gilbert, and Calvin Newport. Smoothed analysis of dynamic networks. *Distributed Computing*, 31(4):273–287, 2018.

23 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.

24 Moez Draief and Milan Vojnovic. Convergence speed of binary interval consensus. *SIAM J. Control. Optim.*, 50(3):1087–1109, 2012.

25 M. J. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *International Conference on Principles of Distributed Systems*, pages 395–409, 2006. doi:10.1007/11945529_28.

26 Leszek Gąsieniec and Grzegorz Stachowiak. Enhanced phase clocks, population protocols, and fast space optimal leader election. *Journal of the ACM (JACM)*, 68(1):1–21, 2020.

27 Leszek Gąsieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 93–102. ACM, 2019.

28 Uri Meir, Ami Paz, and Gregory Schwartzman. Models of smoothing in dynamic networks. In *DISC*, volume 179 of *LIPIcs*, pages 36:1–36:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

29 Othon Michail, Paul G Spirakis, and Michail Theofilatos. Simple and fast approximate counting and leader election in populations. In *Proceedings of the 20th International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 154–169, 2018.

30 Anisur Rahaman Molla and Disha Shur. Smoothed analysis of leader election in distributed networks. In *SSS*, volume 12514 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2020.

31 Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *INFOCOM*, pages 2527–2535. IEEE, 2009.

32 Ryoya Sadano, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. A population protocol model with interaction probability considering speeds of agents. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 2113–2122. IEEE, 2019.

33 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

34 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis: an attempt to explain the behavior of algorithms in practice. *Commun. ACM*, 52(10):76–84, 2009.

35 Yuichi Sudo, Ryota Eguchi, Taisuke Izumi, and Toshimitsu Masuzawa. Time-optimal loosely-stabilizing leader election in population protocols. *arXiv preprint*, 2020. arXiv:2005.09944.

36 Yuichi Sudo and Toshimitsu Masuzawa. Leader election requires logarithmic time in population protocols. *Parallel Processing Letters*, 30(01):2050005, 2020.

37 Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu. Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theoretical Computer Science*, 444:100–112, 2012.

38 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Time-optimal leader election in population protocols. *IEEE Trans. Parallel Distributed Syst.*, 31(11):2620–2632, 2020.

39 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. Loosely-stabilizing leader election with polylogarithmic convergence time. *Theoretical Computer Science*, 806:617–631, 2020.

40 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Self-stabilizing population protocols with global knowledge. *IEEE Transactions on Parallel and Distributed Systems*, (Early Access):1–13, 2021. doi:10.1109/TPDS.2021.3076769.

41 Chuan Xu, Joffroy Beauquier, Janna Burman, Shay Kutten, and Thomas Nowak. Data collection in population protocols with non-uniformly random scheduler. *Theor. Comput. Sci.*, 806:516–530, 2020.

42 Daisuke Yokota, Yuichi Sudo, and Toshimitsu Masuzawa. Time-optimal self-stabilizing leader election on rings in population protocols. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 301–316. Springer, 2020.

A Random initiator-responder order

We show how to extend our proof for the case where the initiator-responder order is random, and no random coins are available. First we change our phase clock algorithm to work without coin flips. The pseudo-code is given as Algorithm 3. It is essentially the same algorithm, but now the initiator increases its *second* value, and the responder sets it to 0. Throughout the rest of the analysis we still refer to "coin flips" made by the agents, where we mean that an agent flips heads if it is an initiator, and tails otherwise. The most important difference to keep in mind is that, while the coin flips for each agent are independent of each other, coin flips between *different agents* are no longer independent.

Algorithm 3 Phase clock.

```
$M \leftarrow \Theta(\log n), S \leftarrow \log(n/p) + \log\log n + O(1)$
$\forall v \in V, v.second \leftarrow 0, v.minute \leftarrow 0, v.hour \leftarrow 0$
foreach interaction $(u, v)$ do
    $u.second \leftarrow u.second + 1$
    $v.second \leftarrow 0$
    if $u.second = S$ then $u.minute \leftarrow u.minute + 1$ and $u.second \leftarrow 0$
    if $u.minute = M$ then $u.hour \leftarrow u.hour + 1$ and $u.minute \leftarrow 0$
    //One-way epidemic
    if $u.hour < v.hour$ then $u.hour \leftarrow v.hour$ and $u.minute \leftarrow 0$ and $u.second \leftarrow 0$
    if $u.hour = v.hour$ and $u.minute < v.minute$ then $u.minute \leftarrow v.minute$ and
      $u.second \leftarrow 0$
```

We now restate the relevant Lemmas. First note that the lower bound on $L(i)$ still holds as we did not assume independence between coin flips made by different agents in the proof of Lemma 5 and Lemma 4. As for the upper bound we did assume independence in Lemma 6, but not in Lemma 7. We state the following alternative for Lemma 6:

▶ **Lemma 10.** *For Algorithm 3, from any configuration where $m' = j < M$ holds, m' increases to $j + 1$ within $O(np^{-2}\log^2 n)$ steps with a constant probability.*

Proof. We maintain the same notations as the proof of Lemma 6, with the exception that we choose $\tau = (Sdp^{-1}\log n)/4$ (larger by an S factor than originally). The proof remains unchanged until we need to bound $Pr[\neg X]$ and $Pr[\neg Y]$. Now there exist dependencies between the coin flips of different agents (but not the coin flips of a single agent).

The proof that $Pr[\neg X]$, remains unchanged. That is, we used a Chernoff bound to state that $Pr[\neg X(v)] < 1/n^2$. Now we have dependencies between coin flips of different agents, however the coin flips of a single agent are still independent. Finally, we note that: $Pr[\neg X] = Pr\left[\bigvee_{v\in V} \neg X(v)\right] \le 1/n$. Where the last transition is due to a union bound.

Next we bound $Pr[\neg Y]$. Again, we expand the expression:

$$\Pr[\neg Y_\tau] = Pr\left[\bigwedge_{v\in V} \neg Y_\tau(v)\right] = Pr\left[\bigwedge_{v\in V}\bigwedge_{i=1}^{\tau} \neg W_i(v)\right] \le Pr\left[\bigwedge_{(v,i)\in U} \neg W_i(v)\right]$$

Our goal is to bound $Pr\left[\bigwedge_{v\in V} \neg Y_\tau(v)\right]$, however there are dependencies between the events. However, it is sufficient if we can find a subset $U \subseteq V \times [\tau]$, such that the events $\{W_i(v)\}_{(v,i)\in U}$ are independent. Note that that every $W_i(v)$ can depend on at most S other events. This is because every round can have length at most S (at which point the round is won). Let us now construct a set U corresponding to *independent* events $\{W_i(u)\}_{(u,i)\in V}$.

This can be constructed greedily, starting with $U = \emptyset, V' = V \times [\tau]$, we add some $(u, i) \in V'$ to U and remove from V' all (v, j) such that event $W_i(u)$ depends on $W_j(v)$. We continue this construction until $V' = \emptyset$. As for every element added to U at most S elements were removed from V', we get that $|U| \geq n\tau/S$. Now we can write:

$$Pr\left[\bigwedge_{(v,i)\in U} \neg W_i(v)\right] \leq (1 - 2^{-S})^{n\tau/S}$$

$$\leq e^{-\frac{(ndp^{-1}\log n)/4}{2^S}} = e^{-\frac{(ndp^{-1}\log n)/4}{2^c np^{-1}\log n}} = e^{-\Omega(d)} \leq 1/3,$$

which completes the proof. ◀

We now state the main theorem for our phase clock.

▶ **Theorem 11.** *Algorithm 1 is a $(np^{-1}\log^2 n,\ np^{-2}\log^3 n)$-phase clock that uses $O((\log n) \cdot (\log(p^{-1}n)))$ states under Γ_s when the initiator-responder order is random.*

Finally let us restate our leader election algorithm for the random initiator-responder order case (Algorithm 4). As before, we can still see this algorithm as flipping coins, but we lose the independence. A slight detail we must notice is that according to our original definition the tick is raised when the agent enters a new epoch. This is now problematic, as when an agent enters a new epoch it is always an initiator, and thus will always increase its level. To overcome this obstacle we can assume that tick is raised one interaction *after* the *hour* variable was increased. This now gives an equal probability for increasing and not increasing the level variable. Our analysis presented in Section 4 does not require independence between coin flips and it goes through unchanged. Thus, we have the following theorem.

Algorithm 4 Leader election.

```
ℓ_max ← Θ(log n)
∀v ∈ V, v.level ← 0
foreach interaction (u, v) do
    if u.level < v.level then  u.leader ← false and u.level ← v.level
    //u.tick ← true one interaction after u enters a new epoch
    if u.tick = true and u.leader = true then
        u.tick ← false
        u.level ← min {u.level + 1, ℓ_max}
    if v.tick = true then  u.tick ← false
    if v.leader = true and u.leader = true then v.leader ← false
```

▶ **Theorem 12.** *For every $p < 1$, Algorithm 4 solves leader election under Γ_s in $O(np^{-2}\log^4 n)$ steps with high probability and in expectation, using $\Theta((\log^2 n) \cdot (\log(n/p)))$ states, when the initiator-responder order is random.*

VBR: Version Based Reclamation

Gali Sheffi ✉
Department of Computer Science, Technion, Haifa, Israel

Maurice Herlihy ✉
Department of Computer Science, Brown University, Providence, RI, USA

Erez Petrank ✉
Department of Computer Science, Technion, Haifa, Israel

Abstract

Safe lock-free memory reclamation is a difficult problem. Existing solutions follow three basic methods (or their combinations): epoch based reclamation, hazard pointers, and optimistic reclamation. Epoch-based methods are fast, but do not guarantee lock-freedom. Hazard pointer solutions are lock-free but typically do not provide high performance. Optimistic methods are lock-free and fast, but previous optimistic methods did not go all the way. While reads were executed optimistically, writes were protected by hazard pointers. In this work we present a new reclamation scheme called *version based reclamation* (VBR), which provides a full optimistic solution to lock-free memory reclamation, obtaining lock-freedom and high efficiency. Speculative execution is known as a fundamental tool for improving performance in various areas of computer science, and indeed evaluation with a lock-free linked-list, hash-table and skip-list shows that VBR outperforms state-of-the-art existing solutions.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages; Software and its engineering → Concurrent programming structures; Theory of computation → Parallel computing models; Theory of computation → Concurrent algorithms; Software and its engineering → Garbage collection; Software and its engineering → Multithreading

Keywords and phrases Safe memory reclamation, concurrency, linearizability, lock-freedom

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.35

Related Version *Full Version*: `https://arxiv.org/abs/2107.13843` [48]

Funding This work is supported by the United States – Israel BSF grant No. 2018655.

1 Introduction

Lock-freedom guarantees eventual system-wide progress, regardless of the behavior of the executing threads. Achieving this desirable progress guarantee in practice requires a lock-free memory reclamation mechanism. Otherwise, available memory space may be exhausted and the executing threads may be indefinitely blocked while attempting to allocate, foiling any progress guarantee. Automatic garbage collection could solve this problem for high-level managed languages, but while some efforts have been put into designing a garbage collector that supports lock-free executions [28, 30, 42–44], a lock-free garbage collector for the entire heap is not available in the literature. Consequently, lock-free implementations must use manual memory reclamation schemes.

Manual reclamation methods rely on *retire* invocations by the program, announcing that a certain object has been unlinked from a data-structure. After an object is retired, the task of the memory reclamation mechanism is to decide when it is safe to reclaim it, making its memory space available for reuse in future allocations. The memory reclamation mechanism ensures that an object is not freed by one thread, while another thread is still using it. Accessing a memory address which is no longer valid may result in unexpected and undesirable program behavior. Conservative reclamation methods make sure that no thread accesses reclaimed space, while optimistic methods allow threads to speculatively access reclaimed memory, taking care to preserve correctness nevertheless.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 35; pp. 35:1–35:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conservative manual reclamation schemes can be roughly classified as either *epoch–based* or *pointer–based.* In epoch–based reclamation (EBR) schemes [9, 22, 23, 33], all threads share a global epoch counter, which is incremented periodically. Additionally, the threads share an announcements array, in which they record the last seen epoch. During reclamation, only objects that had been retired before the earliest recorded epoch are reclaimed. These schemes are often fast, but they are not robust. I.e., a stalled thread may prevent the reclamation of an unbounded number of retired objects. At a worst-case scenario, these objects will consume too much memory space resulting in blocking all new allocations and consequently, foiling system-wide progress.

Pointer–based reclamation methods [26, 35, 37, 50] allow threads to protect specific objects. Namely, before accessing an object, a thread can announce its access in order to prevent this object from being reclaimed. While pointer-based methods can guarantee robustness (and consequentially, lock-freedom), they incur significant time overheads because they need to protect each dereference to shared memory, and issue an expensive memory synchronization fence to make sure the protection is visible to all threads before accessing the derefeneced object. Furthermore, pointer-based schemes are not applicable to many concurrent data-structures (e.g., to Harris's linked-list [22]).

Hybrid schemes that enjoy stronger progress guarantees and smaller time overheads have been proposed. Some epoch-based algorithms try to minimize the chance a non-cooperative thread will block the entire system, by allowing reclamation of objects whose life cycle do not overlap with the activity of the non-responsive thread, e.g., HE (Hazard Eras [45]) and IBR (Interval-Based Reclamation [56]). However, while these algorithms are effectively non-blocking in most practical scenarios, a halted thread can still prevent the reclamation of a large space that relates to the size of the heap. There exists a wait-free variant of the Hazard Eras algorithm [41] which provides a better guarantee but is naturally slower. Another hybrid approach, called *PEBR* [31], obtains lock-freedom at a lower cost, but relies on the elimination of the costly memory fences using the mechanism of Dice et al. [17], which in turn relies on hardware modifications or on undocumented operating systems assumptions that might not hold in the future. Another hybrid of pointer- and epoch-based reclamation is the DEBRA+ and the subsequent NBR [9, 49] reclamation schemes. In order to deal with stuck threads in the epoch-based reclamation, these schemes signal non-cooperative threads and prevent them from postponing the reclamation procedure. While DEBRA+ and NBR are fast, their lock-free property relies on the system's lock-free implementation of signaling. This assumption is not currently available in most existing operating systems, but it may become available in the future.

The first optimistic approach to lock-free memory reclamation was the *Optimistic Access* scheme [13] (also denoted as OA), speculatively allowing reads from retired objects, but protecting writes from modifying retired objects through the use of conservative pointer-based reclamation. After each read, designated per-thread flags signify whether reclamation took place. This allows the reading thread to avoid using stale data loaded from reclaimed space. Subsequent work [11, 12] increased automation and applicability. While the optimistic access reclamation scheme initiated speculative memory reclamation, its mechanism only allowed speculative read instructions. Write instructions were still applied conservatively, using hazard pointers (also denoted as HP) protection to avoid writes on reclaimed space, limiting the benefits of speculative execution.

In this paper we present *VBR*, a novel optimistic memory reclamation scheme that allows full speculative execution, achieving safe and highly efficient lock-free memory reclamation. Both read and write instructions are allowed to access reclaimed space. VBR uses a global epoch counter to assign versions to objects and to (mutable) fields. The versioning of objects ensures that read and write accesses to reclaimed space are guarded from affecting program semantics. The key invariant is that the global epoch counter is guaranteed to increment (at least once) between the time an object is retired and the time its space is re-allocated as a new object. Each logical object is associated with its birth epoch and (eventually) its retire epoch, and each of its mutable fields is associated with a version (representing an epoch smaller or equal to its last update). A speculative read access prudently backs out and retries if the global epoch counter advances while the data structure operation is active. A speculative write operation always fails to modify a re-allocated object due to the modified versions of the object's mutable fields. To support this, each mutable field within a node is represented as a ⟨value, epoch#⟩ pair, and modified only with a double-width compare and swap.

VBR is fully optimistic. Unlike OA [11–13], writes are also speculative and do not require costly fences. Memory fences are used infrequently, upon updating the global epoch. VBR provides full lock-freedom and it does not allow a non-cooperative thread to stall the reclamation process. In fact, VBR never prevents the reclamation of any retired memory object. VBR does not rely on hardware or operating system assumptions (or modifications), except for the existence of a double-word compare and swap instruction, which is available on most existing architectures (e.g., x86).

The proposed VBR can reuse any retired object immediately after it is retired without jeopardizing correctness, which makes it highly space efficient. However, VBR does encounter an issue that pops up in several other schemes [9, 12, 13, 37, 49]. The memory manager sometimes causes a read or a write instruction to "fail" due to a memory reclamation validation test. This failure is not part of the original program control flow and thus, an adequate handling of such failure should be added. In [37] Michael proposes informally to "skip over the hazards and follow the path of the target algorithm when conflict is detected, i.e., try again, backoff, exit loop, etc.". Indeed in many known lock-free data structures [19, 20, 22, 32, 34, 38, 40, 46], handling validation failures is easy, which makes the use of VBR, and other reclamation schemes easy in practice. However, the question arises whether there is a methodology to handle failed validations for all data structures, even when we do not master their specific algorithm. A first rigorous treatment of these failures was provided in [13] for data structures that are written in the normalized form of lock-free data structures [53]. Subsequently, a weaker version of normalized concurrent data structures was presented in [49], where the failure problem is somewhat more severe, as signals may occur at an arbitrary point in the program flow. In this paper, we follow this line of work, and provide a rigorous treatment of a failed read or write validation.

We have implemented VBR on a linked-list, a skip-list, and a hash table and evaluated it against epoch-based reclamation, hazard pointers, hazard eras, interval-based reclamation, and no reclamation at all. As expected, speculative execution outperforms conservative approaches and so VBR yields both lock-freedom as well as high performance.

This paper is organized as follows. In Section 2 we provide an overview of the VBR scheme. In Section 3, we describe our shared memory model and specify some assumptions a data structure must satisfy in order to be correctly integrated with our reclamation mechanism. We describe the VBR scheme integration in Section 4. Experiments appear in Section 5. Related work is surveyed in Section 6. A full correctness proof, along with an illustration of integrating VBR into a lock-free data-structure (Harris's linked-list [22]), appear in the full version of this paper [48].

2 Overview of VBR

The VBR memory reclamation scheme follows an optimistic approach where access to reclaimed objects is allowed. Optimistic approaches reduce the overhead but require care to guarantee correctness. First, VBR allows immediate reclamation of each retired object. There is no need to wait for guards to be lowered to make sure an object is reclaimable as in other methods. This property ensures that stalled threads do not delay reclamation of any object. Second, on strongly-ordered systems (e.g., x86, SPARC TSO, etc.) VBR does not require a costly overhead on read or write accesses. No additional shared memory writes or memory synchronization fences are required with reads or writes to shared memory. This provides the high efficiency seen in the evaluation. However, on weakly-ordered systems (e.g., ARM, PowerPc, etc.), reads must be ordered using special CPU load or memory fence instructions [15]. VBR requires a type-preserving allocator. I.e., a memory space allocated for a specific type is used only for the same type, even when re-allocated. The assumption of type preserving (see also [12, 13, 56]) is necessary for applying our scheme, and is reasonable because data structure nodes are typically fixed-size nodes. Retired nodes are not returned to the operating system. Instead, they are returned to a shared pool of nodes, from which they can be re-allocated by any thread. As in [9, 49], a collection of local node pools (one per thread) is added to the shared pool. A thread accesses the shared pool only when it has no available nodes in its local pool.

Similarly to epoch-based reclamation, VBR maintains a global epoch counter, and as in [3, 41, 45, 56], VBR tracks the birth epoch and retire epoch of each allocated node. The birth epoch is determined upon allocation, and the retire epoch is set upon retirement. The reclamation of a retired node does not involve any action. Upon an allocation of a node, a thread makes sure that the retire epoch of the node is smaller than the current global epoch. If it is not, then the thread increments the global epoch. This ensures that an object is allocated at a global epoch that is strictly larger than its previous retire epoch. Next, the thread re-allocates the object by updating its birth epoch with the current global epoch. This method guarantees that the ABA problem [36] can only occur when the global epoch changes. Namely, when a thread encounters a node during a data-structure traversal, it is guaranteed that this node has not been re-allocated during the traversal if the global epoch has not changed.

VBR allows accessing reclaimed objects, while conservatively identifying reads that may access reclaimed nodes. To identify the access to a reclaimed node, each executing thread keeps track of the global epoch, by reading it upon most shared memory reads (as long as the epoch does not change, this read is likely to hit in the cache). When the thread observes an epoch change, it conservatively assumes that a value was read from a reclaimed memory and it applies a roll-back mechanism (described in Section 4.2), returning to a pre-defined checkpoint in its code. Since a node is always re-allocated at an epoch that is strictly larger than its former retirement, threads never rely on the content of stale values.

We now move on to handling optimistic writes. In addition to the birth epoch and retire epoch, each mutable field (e.g., node pointers) is associated with a version that resides next to it on the data structure node. During the execution, mutable fields are always updated atomically with their associated versions (using a wide CAS instruction). Throughout the life-cycle of a not-yet retired node, all of its versions remain greater than or equal to its birth epoch, and they never exceed its future retire epoch. Versions are decreased or increased during the execution in the following manner: when updating a pointer from a node n to a node m, the pointer's version is set to the maximum birth epoch of the two nodes (either n's or m's). Notice that we assume that n's pointer is never updated after its retirement (for more details, see Section 3.3), and therefore, none of its pointers are assigned a version that exceeds their retirement epoch.

Let us consider the ABA problem for this versioning scheme. The concern is that re-allocations may result in an erroneous success of CAS executions. For example, suppose that a node n points to another node, m, which in turn points to a third node, k. Now, suppose that a thread T_1 tries to remove m by setting n's pointer to point to k. Right before executing the removing CAS, T_1 is halted. While T_1 is idle, T_2 removes m and then reallocates m's space as a new node d. Next, T_2 inserts d as a new node between n and k. In the lack of versions, T_1's CAS will now be erroneously successful. However, with versions it must fail. Since d's birth epoch is necessarily bigger than m's retire epoch, the version in the original pointer to m must be smaller than the version assigned to the pointer when d becomes its referent, and the CAS fails (for more details, see [48]).

3 Settings and Assumptions

In this section we describe our shared memory model and specify the assumptions a data structure must satisfy for integrating with our reclamation mechanism.

3.1 System Model

We use the basic asynchronous shared memory model, as described in [24]. In this model, a fixed set of threads communicate through memory access operations. Threads may be arbitrarily delayed or may crash in the middle of their execution (which immediately halts their execution). The shared memory is accessed via atomic instructions, provided by the hardware. Such instructions may be atomic reads and writes, the compare-and-swap (CAS) instruction and the wide-compare-and-swap (WCAS, which atomically updates two adjacent memory words, and is often supported in commodity hardware [57]) instruction. The CAS operation receives three input arguments: an address of a certain word in memory, an expected value and a new value (both of the size of a single word). It then atomically compares the memory address content to the expected value, and if they are equal, it replaces it with the new received value. Otherwise, it does nothing. The WCAS operation operates in the same manner, on two adjacent memory words.

Concurrent implementations provide different progress guarantees. *Lock-freedom* guarantees that as long as at least one thread executes its algorithm long enough, some thread will eventually make progress (e.g., complete an operation). This progress guarantee is not affected by the scheduler or even by the crash of all threads except for one. For a lock-free data structure to be truly lock-free, it must rely on an allocation method that is also lock-free, because otherwise a blocked allocation can prevent all threads from making progress.

A *data structure* represents a set of *items*, which are distinguished by unique keys, and are often arranged in some order. Each item is represented by a *node*, consisting of both mutable and immutable fields. In particular, each node has an immutable *key* field. The data-structure has a fixed set of *entry points* (e.g., the head of the linked-list in [22]), which are node pointers. A data structure provides the user with a set of operations for accessing it. Moreover, The user cannot access the data-structure in other ways, and the data structure operations never return a node reference. An item that belongs to the data structure set of items must be represented by a node which is reachable from an entry point, by following a finite set of pointers. In particular, the data-structure nodes are accessible only via the entry points. However, a reachable node does not necessarily represent an item in the data structure set of items. We denote the removal of an item from the set of items that the data structure represent by *logical deletion*, and we denote the unlinking of a node from the data structure (i.e., making the node unreachable from the entry points) as *physical deletion.* E.g., in [22], a special mechanism is used in order to mark reachable nodes as deleted. Once a node is marked, it stops representing an item in the data structure set of items (i.e., it is logically deleted), even though it is reachable from an entry point.

3.2 Executions, Histories and Linearizability

A *step* can either be a shared-memory access by a thread (including the access input and output values), a local step that updates its own local variables, an invocation of an operation or the return from an operation (including the respective inputs and outputs). We assume each step is atomic, so an *execution* $E = s_1 \cdot s_2 \cdot \ldots$ consists of a sequence of steps, assumed to start after an initial state, in which all data-structures are initialized and empty. Given E, we further denote the finite sub-execution $s_1 \cdot s_2 \cdot \ldots \cdot s_i$ as E_i.

We follow [29], and model an execution E by its *history* H (and E_i by H_i, respectively), which is the sub-sequence of operation invocation and response steps. A history is *sequential* if it begins with an invocation step, and all invocations (except possibly the last one) have immediate matching responses. We assume that a concurrent system is associated with a *sequential specification*, which is a prefix-closed set of all of its possible sequential histories. A sequential history is *legal* iff it belongs to the sequential specification. An invocation is *pending* in a given history if the history does not contain its matching response. Given a history H, its sub-sequence excluding all pending invocations is denoted as complete(H). An *extension* of a history H is a history constructed by appending responses to zero or more pending invocations in H. We further extend the notion of extensions, and say that an execution E' is an *extension* of an execution E if E is a prefix of E'. In addition, given an execution E, EXT(E) is the set of all histories H' such that (1) H' is an extension of E's respective history, and (2) H' is the respective history of an extension of E. Given a history H and a thread T, T's *sub-history*, denoted as H|T, is the sub-sequence of H consisting of all (and exactly) the steps executed by T. Two histories H and H' are *equivalent* if for every thread T, H|T and H'|T are equal. A history H is *well-formed* if for every executing thread T, H|T is a sequential history. A well-formed history H is linearizable if it has an extension H' for which there exists a legal sequential history S such that (1) complete(H') is equivalent to S, and (2) if a response step precedes an invocation step in H, then it also precedes it in S.

3.3 Implementation Assumptions

We focus on adding the VBR reclamation scheme to lock-free linearizable concurrent data-structure implementations. As in [55], we first assume that modifications are executed using the CAS instruction. No simple writes are used, and no other atomic instructions are supported. Consequently, our scheme does not support the use of other atomic primitives (such as $fetch\&add$ and $swap$).

▶ **Assumption 1.** *All updates occur only via CAS executions.*

In general, as in [45], we assume that all mutable fields of a removed node are invalidated, in order to prevent their future updates. It can be achieved either by marking them [22, 34] or by self-linking (in the case of pointers). More formally:

▶ **Assumption 2.** *Node fields are invalidated (and become immutable) using a designated* invalidate() *method. This method receives as input a node field and invalidates it. The invalidation succeeds iff the field is valid and is not concurrently being updated by another* *thread. In order to check whether a certain field is invalid, a thread calls a designated* isValid() *method. Finally, given a node field, a thread separates the value from the (possible) invalidation mark by calling a designated* getField() *method.*

Following the standard interface for manual reclamation [13, 31, 37, 45, 56], applying our reclamation scheme to an existing implementation includes allocating nodes using an *alloc* instruction and retiring nodes using a *retire* instruction. Nodes are always retired before they can be reclaimed by the reclamation scheme. We assume that it is possible to retire each node only once. To sum up, in a similar way to [37]:

▶ **Assumption 3.** *We assume the following life-cycle of a node n:*

1. ***Allocated**: n is allocated by an executing thread, but is not yet reachable from the data-structure entry points. Once it is physically inserted into the data-structure, it becomes reachable.*
2. ***Reachable** (optional): n is reachable from the data structure entry points, but is not yet necessarily logically inserted into the data-structure (e.g., [47, 52]). When it is made logically included in the data structure it becomes Reachable and valid.*
3. ***Reachable and valid**: n is reachable from the entry points and is considered logically in the data-structure (i.e., valid). At the end of this phase, fields of n are invalidated. We think of n as invalid when at least one of its mutable fields is invalid.*
4. ***Invalid**: n is logically deleted by a designated invalidation procedure, and all of its mutable fields are invalidated (e.g., by marking [22]). Once a field is invalidated, it becomes immutable. At the end of this phase, n is unlinked (physically deleted) from the data structure.*
5. ***Unlinked**: At this point, n is not reachable from the data-structure entry points, and therefore, it is not reachable from any other linked node. At the end of this phase it is retired by some thread. We assume a node is retired only once in an execution. After being retired the node is never linked back into the data structure.*
6. ***Retired**: n has been retired, by a certain thread. We assume that only unlinked nodes can be retired.*

Notice that, as discussed in [12, 20], a node can be physically removed and re-inserted into the data-structure several times during stage 4. However, a *retire* instruction is issued on a node only after it is physically removed for the last time.

Finally, we assume that a thread does not use data on nodes without occasionally checking that the nodes are valid. For our scheme to work, we require this check after modifying the data structure. We assume that if a thread performs a successful modification of the data structure, and if it has some locally saved pointers that were read prior to the modification, then the thread makes limited use of these pointers. Actually, we do not even need to impose the restriction on all modifications. Restrictions are needed only for "important" modifications that cannot be rolled back. Such modifications are called *rollback-unsafe* and they are formally defined in Section 4.2.1 below. In particular:

▶ **Assumption 4.** *If thread T executes any rollback-unsafe modification after updating a local pointer p, then a future use of p is limited. Suppose p references a node n, then a future (i.e., after the rollback-unsafe modification) read of one of n's mutable fields by T is allowed only if the read is followed by an* isValid() *call, and if it returns FALSE, the field content is not used by T.*

While "not using" the content of a read field is intuitively clear, let us also formally say that the content of a read field is not used by a thread T, if T's behavior is indistinguishable from its behavior when reading the $\perp$ sign instead of the actual value read. Note that even after a rollback-unsafe update, T is allowed to use the content of fields that were read before the modification. However, after the modification, T is not allowed to dereference a local pointer and read values from the referenced node without checking the validity of the node. For example, T is allowed to use previously read pointers as expected values of a CAS, or as the target of a write operation. T can traverse a list in a wait free manner (since there is no modification involved). T can trim all invalid nodes along a traversal. This is allowed since trimming includes checking the validity of the traversed nodes. All known lock-free data structures that we are aware of (e.g., [19, 20, 22, 27, 32, 34, 38, 40, 46]) satisfy Assumption 4.

4 VBR: Version Based Reclamation

In this section we present VBR: a lock-free recycling support for lock-free linearizable [29] data-structures. We start by describing the reclamation scheme and the modifications applied to the nodes' representation in Section 4.1, and continue with the modifications applied to the data-structure operations in Section 4.2. In Section 4.2.1 we define the notion of code checkpoints and show how to insert them into an existing linearizable implementation. In Section 4.2.2 we go over the necessary adjustments to read operations (from shared variables). Handling update operations is described in Section 4.2.3. A full example API appears in Figure 1. For ease of presentation, we refer to data-structures for which each node has a single immutable field (the node's key) and a single mutable field (the node's next pointer), and nodes' invalidation is executed via the marking mechanism [22]. However, this interface can be easily extended to handle multiple immutable and mutable fields, and other invalidation schemes. We present a full correctness proof for Theorem 1 in [48].

▶ **Theorem 1.** *Given a lock-free linearizable data-structure implementation, satisfying all of the assumptions presented in Section 3.3, it remains lock-free and linearizable after integrating it with VBR according to the modifications described in Sections 4.1–4.2.*

4.1 The Reclamation Mechanism

VBR uses a shared epoch counter, denoted e, incremented periodically by the executing threads. In addition, each executing thread keeps track of the global epoch using a local *my_e* variable. Each node is associated with *birth_epoch* and *retire_epoch* fields. Its birth epoch contains the epoch seen by the allocating thread upon its allocation, and its retire epoch contains the epoch seen by the thread which removed this node from the data structure, right before its retirement. We add a version field adjacent to each mutable field (e.g., node pointers). The field's version is guaranteed to always be equal to or greater than the node's birth epoch, and equal to or smaller than its eventual retire epoch (if there exists any). The field's data and its associated version are always updated together. E.g., see lines 9, 33.

Handling reclamation at the operating system level often requires using locks (unless it is configured to ignore certain traps). Therefore, for maintaining lock-freedom, VBR uses a user-level allocator. Retired nodes are inserted into manually-managed node pools [25, 54] for future re-allocation. We use a type-preserving allocator. I.e., a memory space allocated for a specific type is used only for the same type, even when re-allocated.

Each thread's allocation and reclamation mechanism works as follows. Besides sharing a global nodes pool [25, 54], each executing thread maintains a local pool of retired nodes, from which it retrieves reclaimed nodes for re-allocations. When the local pool becomes large enough, retired nodes may be moved to a global pool of retired nodes, allowing re-distribution of reclaimed nodes between the threads. When retiring a node, it is possible to re-allocate this node immediately. However, we use a local retired list to stall its re-allocation for a while. This allows infrequent increments to the global epoch counter, which improves performance. A retired node is therefore added to the private list of retired nodes. When the size of the retired list exceeds a pre-defined threshold, it is appended as a whole to the thread's local allocation pool, becoming available for allocation.

The full allocation method appears in lines 1–11 of Figure 1. First, the thread reads the retire epoch of the next available node in its allocation pool. If it is equal to the shared epoch counter, then the thread increments the shared epoch counter using CAS (line 4) and executes a rollback to the previous checkpoint (for more details, see Section 4.2.1). This makes sure that the birth epoch of a new node is larger than the retire epoch of the node that was previously allocated on the same memory space. If the CAS is unsuccessful, then another thread has incremented the global epoch value and there is no need to try incrementing it again. If e is bigger than the retired node's retire epoch, the thread sets the new node's birth epoch to its current value. After setting the node's birth epoch, its next pointer version is set to this value, along with an initialization of its data to NULL in line 9. Due to Assumption 3 (the mutable fields of a node become immutable before it is retired), the WCAS executed in line 9 is always successful. Finally, the key field is set to the key received as input.

The *retire* method appears in lines 12–16. First, the retiring thread makes sure that the node is not already retired in line 13 (for more details, see [48]). Then, it sets the node's retire epoch to be the current global epoch, and appends the retired node to its local retired nodes list. In case its local copy of the global epoch counter is not up to date, it performs a checkpoint rollback in line 16 (for more details, see Section 4.2.1).

4.2 Code Modifications

Unlike former reclamation methods, VBR allows both optimistic reads and optimistic writes. Namely, the executing threads may sometimes access a previously reclaimed node, and either read its stale values or try to update it. To the best of our knowledge, there exists no other scheme which allows optimistic writes, and optimistic reads are allowed only in [11–13]. Our versioning mechanism ensures that a write to a previously reclaimed node always fails, and that stale values that are read from reclaimed nodes are always ignored. This gives rise to an additional problem – when failing to read a fresh value due to an access to a reclaimed node, the program needs to move control to an adequate location. This problem does not arise with epoch based reclamation because a thread never fails due to a test that the memory reclamation scheme imposes. Failures that arise due to optimistic access are not part of the original lock-free concurrent data structure. Interestingly, deciding how to treat failed reads or writes is very easy in practice. We could easily modify lock-free data structures that satisfy the assumptions presented in Section 3.3 (e.g., [19, 20, 22, 32, 34, 38, 40, 46]), at a minimal performance cost. However, while presenting this scheme, we would also like to propose a general manner to handle failed accesses. We are going to define the notion of execution checkpoints. Upon accessing an allegedly stale value, the code just rolls back to the appropriate checkpoint. This problem is given general treatment in the format of a normalized form assumption in [12, 13], and in a total separation between read and write phases during the execution in [49]. Although the VBR scheme can be applied to implementations that adhere to both models, both of them require extensive modifications to the original program's structure. Therefore, we propose a new method, which is more general and makes less assumptions on the given implementation. Our method is to carefully define program checkpoints.

```
alloc(int key)
    n := alloc_list $\rightarrow$ next
    if (n $\rightarrow$ retire_epoch $\geq$ my_e)
        CAS(&e, my_e, my_e $+ 1$)
        alloc_list $\rightarrow$ next := n
        return to checkpoint                                        ▷ Checkpoint rollback
    n $\rightarrow$ birth_epoch := my_e
    n $\rightarrow$ retire_epoch := $\perp$
    WCAS(&(n $\rightarrow$ next), $\langle$n $\rightarrow$ next.data, n $\rightarrow$ next.version$\rangle$, $\langle$NULL, my_e$\rangle$)
    n $\rightarrow$ key := key
    return n

retire(Node* n, long n_b)
    if (n $\rightarrow$ birth_epoch $>$ n_b || n $\rightarrow$ retire_epoch $\neq \perp$) return     ▷ Avoiding double retirements
    n $\rightarrow$ retire_epoch := e.get()
    retired_list $\rightarrow$ next := n
    if (n $\rightarrow$ retire_epoch $>$ my_e) return to checkpoint                   ▷ Checkpoint rollback

getNext(Node* n)
    n_next := unmark(n $\rightarrow$ next.data)
    n_next_b := n_next $\rightarrow$ birth_epoch
    if (my_e $\neq$ e.get()) return to checkpoint                               ▷ Checkpoint rollback
    return n_next, n_next_b

getKey(Node* n)
    n_key := n $\rightarrow$ key
    if (my_e $\neq$ e.get())  return to checkpoint                              ▷ Checkpoint rollback
    return n_key

isMarked(Node* n, long n_b)
    res := isMarked(n $\rightarrow$ next.data)
    if (n $\rightarrow$ birth_epoch $\neq$ n_b) return TRUE                        ▷ The node is already removed
    return res

update(Node* n, long n_b, Node* exp, long exp_b, Node* new, long new_b)
    exp_v := max { n_b, exp_b }
    new_v := max { n_b, new_b }
    return WCAS(&(n $\rightarrow$ next), $\langle$ exp, exp_v $\rangle$, $\langle$ new, new_v $\rangle$)

mark(Node* n, long n_b)
    exp := unmark(n $\rightarrow$ next.data)
    exp_v := max { n_b, exp $\rightarrow$ birth_epoch }
    if (n $\rightarrow$ birth_epoch $\neq$ n_b) return FALSE                       ▷ The node is already removed
    new := mark(exp)
    return WCAS(&(n $\rightarrow$ next), $\langle$ exp, exp_v $\rangle$, $\langle$ new, exp_v $\rangle$)
```

Figure 1 An example VBR interface.

4.2.1 Defining Checkpoints

VBR occasionally requires a rollback to a predefined checkpoint. In order to install checkpoints in a given code in an efficient manner, one needs to be able to distinguish important shared-memory accesses that cannot be rolled back from non-important accesses that allow rolling back. The notion of important shared-memory accesses is similar to the definition of an *owner CAS* in [53], and shares some mutual concepts with the *capsules* definition, given in [5, 6]. Informally, non-important shared-memory accesses (that can be rollbacked) either do not affect the shared memory view (e.g., reading from the shared-memory) or do not have any meaningful impact on the execution flow. For example, consider Hariss's implementation of a linked-list [22]. The physical removal of a node, i.e., trimming the node from the list after it has been marked, can be safely rollbacked. If we try the same trim again, it will simply fail, and if we rollback further, this trim will not even be attempted. However, the (successful) insertion of a new node into the list and the (successful) marking of a node for logical deletion are both important and are not rollback-safe. In both cases, performing a rollback right after the successful update would result in a non-linearizable history (as the inserter or remover would not return TRUE after successfully inserting or removing the node, respectively). We now define the notion of *rollback-safe steps* in a given execution E with a respective history H.

▶ **Definition 2** (Rollback-Safe Steps). *We say that s_i is a rollback-safe step in an execution E if $EXT(E_i)=EXT(E_{i-1})$.*

If s_i is not a rollback-safe step, then we say that it is a *rollback-unsafe step*. According to Definition 2, if s_i is a rollback-safe step, executed by a thread T during E, then T can safely perform a *local rollback step* after s_i. I.e., right after executing s_i by T, T can restore the contents of all of its local variables and program counter (assuming they were saved before s_i), and the obtained execution would have the same set of corresponding history extensions. Note that, by Definition 2, local steps, shared memory reads and unsuccessful memory updates (CAS executions returning FALSE) are considered as rollback-safe steps. We extend Definition 2 in the following manner: a thread T can rollback to any previously saved set of local variables (including its program counter), as long as it has not performed any rollback-unsafe steps since they had been saved. I.e., it can safely rollback to its last visited checkpoint.

Given a code for a concurrent data-structure, checkpoints are first installed after some shared memory update instructions, in the following manner: let l be a shared-memory update instruction (i.e., a CAS instruction). If there exists an execution $E = s_1 \cdot \ldots$ such that l is executed successfully during a step s_i (i.e., the CAS execution returns TRUE), and s_i is a rollback-unsafe step in E, then a checkpoint is installed right after l. The installation of a checkpoint includes a check that the update is indeed successful (the CAS execution returns TRUE). If it is, then the checkpoint reference is updated (to the current value of the program counter), and all local variables are saved for a future restoration[1]. If the update is not successful (the CAS execution returns FALSE), then nothing is done. Checkpoints are also installed in the beginning of each data-structure operation. As opposed to the first type of checkpoint triggers, the installation does not depend on anything when done upon an operation invocation. Recall that, by Definition 2, an operation invocation is always a rollback-unsafe step. After rolling back to a checkpoint, the thread updates its local copy of the global epoch, recovers its set of local variables, and continues its execution[2].

4.2.2 Read Methods

As threads may access stale values, the only way to avoid relying on a stale value is to constantly check that the node from which the value was read has not been re-allocated. In a conservative way, we think of a read instruction as potentially reading a stale value if the global epoch number changed since the last checkpoint. A read of a stale value must imply a change of the global epoch number because the birth epoch of a node is strictly larger than the retirement epoch of a previous node that resides on the same memory space. Therefore,

[1] It is unnecessary to save uninitialized variables and variables that are not used anymore

[2] Right before a thread rolls-back to its previous checkpoint, it handles some unlinked nodes for guaranteeing VBR's robustness. As this issue does not affect correctness, we move this discussion to [48].

the shared epoch counter e is read upon each operation invocation (see Section 4.2.1), each node retirement, and after certain allocations and reads from the shared memory. Since a node cannot be allocated during an epoch in which a node, previously allocated from the same memory address, is not yet removed from the data-structure (see lines 3-6 in Figure 1), as long as the global epoch, read before the read of a node, is equal to the one read after the read of the node, it is guaranteed that the node's value is not stale. In general, when reading a node pointer into a local variable, it is always saved together with the node's birth epoch, as the node is represented by its birth epoch as well.

W.l.o.g. and for simplifying our presentation, we assume each node originally consists of an immutable key field and a mutable next pointer field, and that a node is invalidated using the *mark()* method [22]. Therefore, there are roughly three types of read-only accesses in the original reclamation-free algorithm. The first type is the read of a node via the next pointer of its predecessor, the second one is the read of a node's key, and the third one is the read of a node's mark bit. We install the *getNext()* method instead of each pointer read in the original code, the *getKey()* method instead of each key read, and a new *isMarked()* method instead of the original one. Accesses to other (mutable or immutable) node fields should be very similar, and therefore require the same treatment.

The code for the *getNext()* method appears in lines 17-21, and the code for the *getKey()* method appears in lines 22-25. Both methods receive a pointer to the target node (assumed to be given as an unmarked pointer). First, the next node and its birth epoch (or the key, respectively) are saved in local variables. Then, the global epoch is read and compared to the previous recorded epoch. If the epoch has changed since the previous read, then the values may be stale, and the execution returns to the last checkpoint. Otherwise, the data is returned in line 21 (or line 25, respectively).

The code for the VBR-integrated *isMarked()* method appears in lines 26-29. It also receives an unmarked pointer to the target node, and additionally, its birth epoch. It first checks whether the node's next pointer is indeed marked (line 27), using the original *isMarked()* method, which receives the actual allegedly marked pointer and checks if it is marked. Then the node's birth epoch is read, for guaranteeing that the given node is the correct one (and not another one, allocated from the same memory space). If it is not, then the target node has certainly been marked in the past, and the method returns TRUE in line 28. Otherwise, it returns the answer received in line 27. As this method returns a correct answer regardless of epoch changes or the retirement of the target node, it does not read the global epoch nor returns to the last checkpoint.

4.2.3 Update Methods

By Assumption 1, all data structure updates are executed using CAS instructions. We consider two types of pointer updates. The first type, depicted in lines 30-33 of Figure 1, is the update of an unmarked pointer. The second type (lines 34-39) is the marking of an unmarked pointer. The update of other mutable fields can be similarly implemented. In particular, the version of non-pointer mutable fields should always be equal to the node's birth epoch (which makes such fields much easier to handle).

The *update()* method replaces the original pointer update via a single CAS instruction. It receives pointers to the target node, its expected successor and the new successor, together with their respective birth epochs. All three pointers are assumed to be unmarked. The expected and new pointer versions are calculated in the same manner (lines 31-32): the maximum birth epoch of the target node and successor node. The next field is either successfully updated or remains unchanged in line 33. In [48] we prove that the target node's next pointer is updated iff (1) it has not been reclaimed yet, (2) it is not marked, and (3) it indeed points to the expected node (including the given birth epoch).

The *mark()* method marks an unmarked *next* pointer, without changing its pointed node. It receives the target node and its birth epoch. The actual marking is executed in line 39. It uses the unmarked and marked variants of the pointer, and does not change the pointer's version (calculated in line 36). In [48] we prove that the target node is marked iff (1) it has not been reclaimed yet, (2) it is not marked, and (3) it indeed points to the expected node (read in line 35), just before the marking.

5 Evaluation

For evaluating throughput of VBR we implemented lock-free variants of a linked-list from [34], a hash table (implemented using the same list), and a skip list. We implemented Herlihy and Shavit's lock-free skiplist [27] with the amendment suggested in [20] for lock-free reclamation. VBR was integrated into all data-structures according to the guidelines presented in Section 4.

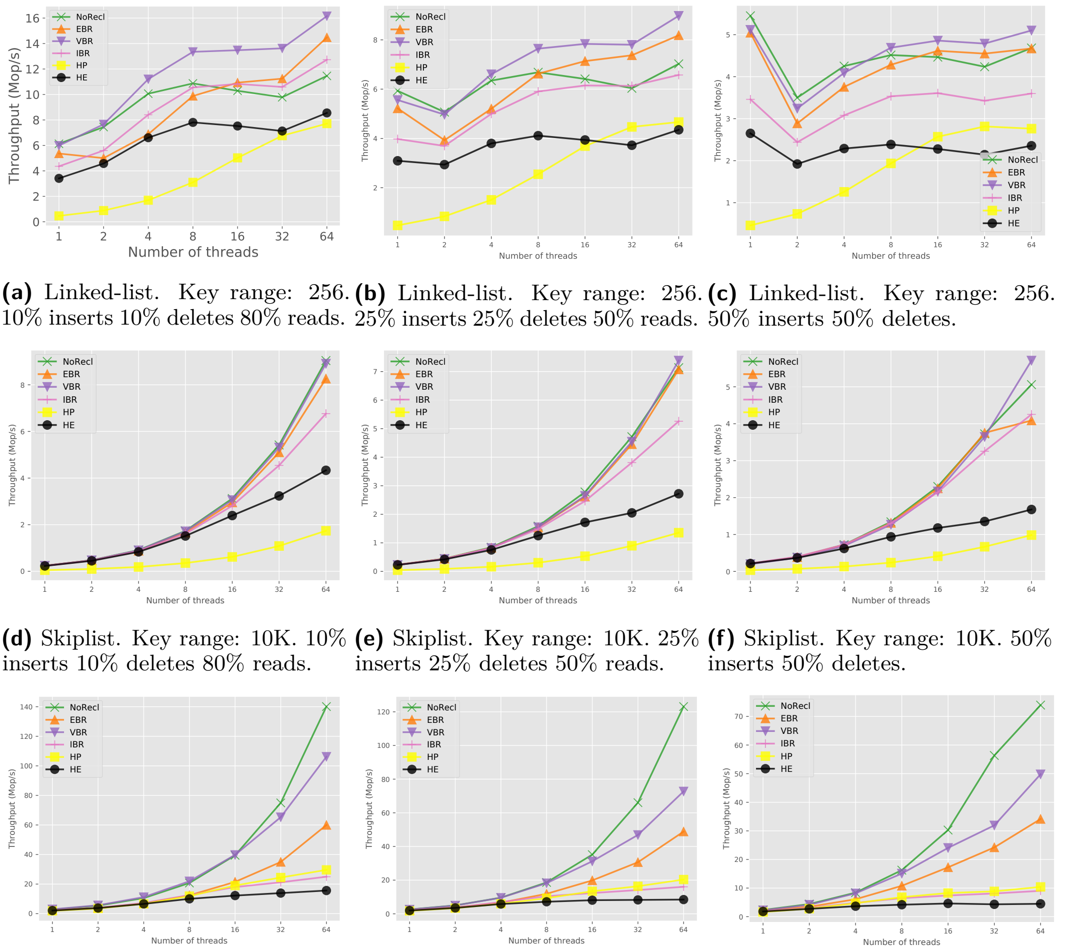
We evaluated VBR against a baseline execution in which memory is never reclaimed (denoted NoRecl), an optimized implementation of the epoch-based reclamation method [22] (denoted EBR), the traditional hazard pointers scheme [37] (denoted HP), the hazard eras scheme [45] (denoted HE), and the 2GEIBR variant of the interval-based scheme [56] (denoted IBR). For all reclamation schemes, we implemented optimized local allocation pools. Objects were reclaimed once the retire list is full, and were allocated from the shared pool if there were no objects available in the local pool. As retired objects cannot be automatically reclaimed in EBR, IBR, HE and HP, we tuned their retire list sizes in order to achieve high performance. We further tuned the global epoch update rate in EBR, HE and IBR (in VBR it seldom happens and does not require any tuning).

Pointer-based methods require that it would not be possible to reach a reclaimed node by traversing the data structure from a protected node, even if the protected node has been unlinked and retired. This prevents schemes like HP, HE, IBR, etc. from being used with some data structures such as Harris's original linked-list [22] or the lock-free binary tree of [10, 40]. We did not implement binary search trees, because some of the measured competing schemes cannot support it.

5.1 Setup

Our experimental evaluation was performed on an Ubuntu 14.04 (kernel version 4.15.0) OS. The machine featured 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores (64 threads overall). The machine used 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores and an L3 cache of 6MB per processor. The code was compiled using the GCC compiler version 7.5.0 with the -O3 optimization flag.

We implemented object pools in a similar way to [12], to avoid returning reclaimed objects to the OS. All schemes used that implementation, in which all pools are pre-allocated before the test. Each test was a fixed-time micro benchmark in which threads randomly call the *Insert()*, *Delete()* and *Search()* operations according to three workload profiles: (1) a search-intensive workload (80% searches, 10% inserts and 10% deletes), (2) a balanced workload (50% searches, 25% inserts and 25% deletes), and (3) an update-intensive workload (50% inserts and 50% deletes). Each execution started by filling the data-structure to half of its range size. For the hash-table, the load factor was 1. We measured the throughput of



(a) Linked-list. Key range: 256. 10% inserts 10% deletes 80% reads.

(b) Linked-list. Key range: 256. 25% inserts 25% deletes 50% reads.

(c) Linked-list. Key range: 256. 50% inserts 50% deletes.

(d) Skiplist. Key range: 10K. 10% inserts 10% deletes 80% reads.

(e) Skiplist. Key range: 10K. 25% inserts 25% deletes 50% reads.

(f) Skiplist. Key range: 10K. 50% inserts 50% deletes.

(g) Hash table. Key range: 10M. 10% inserts 10% deletes 80% reads.

(h) Hash table. Key range: 10M. 25% inserts 25% deletes 50% reads.

(i) Hash table. Key range: 10M. 50% inserts 50% deletes.

Figure 2 Throughput evaluation. Y axis: throughput in million operations per second. X axis: #threads.

the above schemes. Each experiment lasted 1 second (as longer executions showed similar results) and was run with a varying number of executing threads. Each experiment was executed 10 times, and the average throughput across all executions was calculated.

5.2 Discussion

Figure 2 shows that VBR is faster than other manual reclamation schemes, even when contention is high (Figures 2a-2c), and in update-intensive workloads (Figures 2c, 2f, 2i). VBR outperforms epoch-based competitors (EBR, IBR and HE) due to its infrequent epoch updates. In order to avoid allocation bottlenecks, EBR, IBR and HE require frequent epoch updates. I.e., many global epoch accesses result in cache misses and slow down the allocation process, the reclamation process and the operations executions. In contrast to these methods, VBR requires infrequent epoch updates. An increment is triggered when the next node to be allocated has a retire epoch equal to the current global epoch. Most global epoch accesses during VBR hit the cache, and are negligible in terms of performance.

In addition, VBR outperforms its pointer-based competitors (IBR, HE and HP) since it requires neither read nor write fences. Specifically, in the hash table implementation, it surpasses the next best algorithm, EBR, by up to 60% in the search-intensive workload (Figure 2g), by up to 50% in the balanced workload (Figure 2h), and by up to 40% in the update-intensive workload (Figure 2i). In the skiplist implementation, VBR is comparable to the baseline and EBR for the search-intensive and balanced workloads (Figures 2d-2e). For the update-intensive workload, it outperforms the next best algorithm, IBR, by up to 35% (Figure 2f). In the linked-list implementation, VBR outperforms the next best algorithm, EBR, by up to 10%, 11% and 8%, respectively (Figures 2a-2c). For cache locality reasons, VBR outperforms the baseline execution for all linked-list and skiplist workloads and for all key ranges (Figures 2a-2f). As cache locality plays no role in the hash table implementation, VBR has no advantage against the baseline for this data-structure. VBR's throughput is around 75% of the baseline for the search-intensive workload, around 60% of the baseline for the balanced workload and around 65% of the baseline for the update-intensive workload.

6 Related Work

Much related work was already discussed in the introduction. There are many memory management schemes, and in the evaluation we compared VBR against highly efficient schemes whose code is available (We could not compare against all). Previous works [31, 49] defined a set of desirable reclamation properties. Safe reclamation algorithms should be fast (show low latency and high throughput), robust (the number of unreclaimed objects should be bounded), widely applicable and self-contained (not relying on external features).

Two novel methods initiated the study of memory reclamation for concurrent data structures. Pointer-based schemes [17, 26, 37] protect objects that are currently accessed by placing a hazard pointer referencing them. These methods are often slow and not always applicable. Epoch-based schemes [9, 22] (and quiescent state based schemes [23]) wait until all threads move to the next operation to make sure that an unlinked node cannot be further accessed. Such methods are sometimes not robust, and most hybrids of the two approaches [4, 9, 41, 45, 56] are either not always applicable, or rely on special hardware support. Drop-the-anchor [8] extends HP by protecting only some of the traversed nodes and reclaiming carefully, yet it is not easily applicable and has only been applied to linked-lists.

Another approach, which is neither fast nor robust, is reference counting based reclamation [7, 16, 21, 26]. This scheme keeps an explicitly count of the number of pointers to each object, and reclaims an object with a zero count. Such schemes require a way to break cyclic structures of retired objects, and are often slow or rely on hardware assumptions. This scheme has a wait-free (and in particular, robust) variant [51] and a lock-free variant [14], but they are not fast, since they require multiple expensive synchronization fences.

Many schemes rely on Hardware-specific or OS features, or affect the execution environment. ThreadScan [2], StackTrack [1], and Dragojević et al. [18] rely on transactional memory for the reclamation, which is not always available in hardware or may be slow in a software implementation. DEBRA+ [9] and NBR [49] use OS signals in order to wake unresponsive threads and allow lock-free progress even for EBR-based methods, if the OS signal implementration is lock-free. Morrison and Afek [39] avoid memory fences by waiting for a short while. This relies on specific hardware properties that might not always be available. Dice et. al. [17] and PEBR [31] avoid costly fences by relying on the existence of process-wide memory fences. QSense [4] requires control of the OS scheduler. In particular, to make hazard pointers visible, threads are periodically swapped out.

References

1 Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

2 Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. *ACM Transactions on Parallel Computing (TOPC)*, 4(4):1–18, 2018.

3 Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.

4 Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 349–359, 2016.

5 Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2019.

6 Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 247–258, 2018.

7 Guy E Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time. *arXiv preprint*, 2020. `arXiv:2002.07053`.

8 Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 33–42, 2013.

9 Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

10 Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47(4):199–210, 2012.

11 Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proc. ACM Program. Lang.*, 2(OOPSLA):143:1–143:24, 2018. `doi:10.1145/3276513`.

12 Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. *ACM SIGPLAN Notices*, 50(10):260–279, 2015.

13 Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 254–263, 2015.

14 Andreia Correia, Pedro Ramalhete, and Pascal Felber. Orcgc: automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 205–218, 2021.

15 std::memory_order. `https://en.cppreference.com/w/cpp/atomic/memory_order`. Accessed: 2021-04-19.

16 David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

17 Dave Dice, Maurice Herlihy, and Alex Kogan. Fast non-intrusive memory reclamation for highly-concurrent data structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, pages 36–45, 2016.

18 Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 99–108, 2011.

19 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.

20 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

21 Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2008.

22 Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

23 Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

24 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

25 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

26 Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)*, 23(2):146–196, 2005.

27 Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming*. Newnes, 2020.

28 Maurice P Herlihy and J Eliot B Moss. Lock-free garbage collection for multiprocessors. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 229–236, 1991.

29 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

30 Richard L Hudson and J Eliot B Moss. Sapphire: Copying gc without stopping the world. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57, 2001.

31 Jeehoon Kang and Jaehwang Jung. A marriage of pointer-and epoch-based reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 314–328, 2020.

32 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013.

33 Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

34 Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.

35 Maged M Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, 2002.

36 Maged M Michael. Aba prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

37 Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

38 Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

39 Adam Morrison and Yehuda Afek. Temporally bounding tso for fence-free asymmetric synchronization. *ACM SIGARCH Computer Architecture News*, 43(1):45–58, 2015.

40 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

41 Ruslan Nikolaev and Binoy Ravindran. Universal wait-free memory reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 130–143, 2020.

42 Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *Proceedings of the 6th international symposium on Memory management*, pages 159–172, 2007.

43 Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices*, 43(6):33–44, 2008.

44 Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. *ACM Sigplan Notices*, 45(6):146–159, 2010.

45 Pedro Ramalhete and Andreia Correia. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 367–369, 2017.

46 Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

47 Gali Sheffi, Guy Golan-Gueta, and Erez Petrank. A scalable linearizable multi-index table. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 200–211. IEEE, 2018.

48 Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation, 2021. `arXiv:2107.13843`.

49 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–190, 2021.

50 Daniel Solomon and Adam Morrison. Efficiently reclaiming memory in concurrent search data structures while bounding wasted memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 191–204, 2021.

51 Håkan Sundell. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2005.

52 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *International Conference On Principles Of Distributed Systems*, pages 330–344. Springer, 2012.

53 Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. *ACM SIGPLAN Notices*, 49(8):357–368, 2014.

54 R Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

55 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

56 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. Interval-based memory reclamation. *ACM SIGPLAN Notices*, 53(1):1–13, 2018.

57 Pavel Yosifovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more.* Microsoft Press, 2017.

Extension-Based Proofs for Synchronous Message Passing

Yilun Sheng
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China

Faith Ellen
Department of Computer Science, University of Toronto, Canada

Abstract

There is no wait-free algorithm that solves k-set agreement among $n \geq k+1$ processes in asynchronous systems where processes communicate using only registers. However, proofs of this result for $k \geq 2$ are complicated and involve topological reasoning. To explain why such sophisticated arguments are necessary, Alistarh, Aspnes, Ellen, Gelashvili, and Zhu recently introduced extension-based proofs, which generalize valency arguments, and proved that there are no extension-based proofs of this result.

In the synchronous message passing model, k-set agreement is solvable, but there is a lower bound of t rounds for any k-set agreement algorithm among $n > kt$ processes when at most k processes can crash each round. The proof of this result for $k \geq 2$ is also a complicated topological argument. We define a notion of extension-based proofs for this model and we show there are no extension-based proofs that t rounds are necessary for any k-set agreement algorithm among $n = kt + 1$ processes, for $k \geq 2$ and $t > 2$, when at most k processes can crash each round. In particular, our result shows that no valency argument can prove this lower bound.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Theory of computation → Complexity theory and logic; Theory of computation → Distributed algorithms; Theory of computation → Distributed computing models

Keywords and phrases Set agreement, lower bounds, valency arguments

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.36

Acknowledgements Faith Ellen would like to thank Sergio Rajsbaum and Leqi Zhu for some preliminary discussions about this problem. Support is gratefully acknowledged from the Natural Science and Engineering Research Council of Canada under grant RGPIN-2020-04178. Part of this work was done while Yilun Sheng was virtually visiting University of Toronto.

1 Introduction

In the *k-set agreement* problem, each process has an input from $\{0, \ldots, k\}$ and each process that does not crash must output a value from among the inputs *(validity)* such that at most k different values are output *(k-agreement)*. In 1993, Borowsky and Gafni [6], Herlihy and Shavit [11], and Saks and Zaharoglou [14] concurrently proved, using sophisticated topological proofs, that there are no wait-free algorithms that solve k-set agreement among $n \geq k+1$ processes in asynchronous models where processes communicate by reading from and writing to shared registers (or objects that can be built from registers).

Extension-based proofs were recently introduced by Alistarh, Aspnes, Ellen, Gelashvili, and Zhu [2, 3] as a generalization of valency arguments. They proved that, for $k \geq 2$, there are no extension-based proofs of this impossibility result in the iterated immediate snapshot and iterated snapshot models. These are asynchronous shared-memory models that are closely related to the models used by Borowsky and Gafni, Herlihy and Shavit, and Saks and Zaharoglou. This explains the necessity of the topological arguments used to prove this

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 36; pp. 36:1–36:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

impossibility result. Subsequently, Attiya, Castañeda, and Rajsbaum [4] presented a simpler, but more restricted, class of proofs and showed that, for $k \geq 2$, this class contains no proof of the impossibility of a uniform algorithm solving k-set agreement among $n \geq k+1$ processes in the iterated immediate snapshot model.

In the synchronous message passing model, Chaudhuri, Herlihy, Lynch, and Tuttle [7] proved that $\lfloor f/k \rfloor + 1$ rounds are necessary and sufficient to solve k-set agreement among $n \geq k + f + 1$ processes when at most f processes can crash. Their upper bound is a simple, elegant algorithm, which is described in Section 3. The proof of their lower bound is a complicated topological argument, which constructs a k-dimensional simplex of executions in which adjacent executions are indistinguishable to certain processes. Subsequent papers by Herlihy, Rajsbaum, and Tuttle [9, 10] presented more succinct proofs using additional topological tools, showing that it is impossible to solve k-set agreement among $n \geq k(t+1)+1$ processes in t rounds, assuming at most k processes crash each round. Gafni [8] gave a different proof. He presented a technique that converts any synchronous t round k-set agreement algorithm for $n \geq k(t+1)+1$ processes, assuming at most k processes crash each round, into an asynchronous k-set agreement algorithm for $n \geq k+1$ processes, assuming at most k processes crashes. Since there is no such asynchronous algorithm, he obtained the same lower bound.

A natural question is whether there is a simpler proof of this lower bound, analogous to the valency arguments used to prove that more than t rounds are necessary to solve binary consensus among $n \geq t+2$ processes when at most one process can crash each round [1, 5, 13]. Following the approach by Alistarh, Aspnes, Ellen, Gelashvili, and Zhu [2, 3], we define a version of extension-based proofs for the synchronous message passing model in Section 4. Then, in Section 5, we prove that for $k, t \geq 2$, there is no extension-based proof of the impossibility of solving k-set agreement among $n = k(t+1)+1$ processes in t rounds when at most k processes crash each round.

2 Model

A *synchronous message passing model* consists of a set of n processes, $P = \{p_0, p_1, \ldots, p_{n-1}\}$. Executions of an algorithm proceed in synchronous *rounds*, in which each process (that has not already terminated or crashed) sends messages to other processes, then receives the messages other processes sent to it in that round, updates its state based on the messages it receives, and either outputs a value and terminates or proceeds to the next round.

A process *crashes* during some round if there is some other process to which it is supposed to send a message during that round, but it does not. If a process crashes during a round, it does not send any messages in any subsequent round and it does not output a value. A process is *active* at the beginning of round $r \geq 1$ if it has not terminated or crashed during the first $r-1$ rounds. Every process is active at the beginning of round 1.

Initially, each process p_i has an input x_i that is (part of) its state. A *configuration* is an n-component vector C that describes the states of all processes at the beginning of some round. If process p_i is active, then $C[i]$ is its state at the beginning of this round. If it terminated in a previous round, then $C[i] = (\top, m)$, where m is the value it output. If it crashed in a previous round, then $C[i] = \bot$. An *initial configuration* consists of the state of every process before any computation occurs. A *final configuration* is a configuration in which no process is active, i.e., every process has either terminated or crashed.

An *algorithm* specifies what message each active process wants to send to every other process in each round, as a function of its state at the beginning of the round. A *full information algorithm* is an algorithm in which, in every round, each active process p_i sends its state to every other process and updates its state to be the n-component vector s, where $s[i]$ is its previous state and $s[j]$ is the last message it received from process p_j, for all $j \neq i$. If process p_i has never received a message from process p_j, then $s[j] = \perp$. If process p_i terminates in state s and outputs m, then $s = (\top, m)$. Note that every algorithm can be transformed into a full information algorithm using the same number of rounds: It simply does not pay attention to any extra information it has received when deciding when to terminate and what value to output.

A *one round schedule* is an n-component vector σ, where $\sigma[i] \subseteq P$ consists of p_i and the set of processes to which p_i sent messages during the round, if p_i is active at the beginning of the round, and is $\emptyset$, otherwise. The one round schedule σ is *applicable* to configuration C if $p_i \in \sigma[i]$ for every process p_i that is active in C and $\sigma[i] = \emptyset$ for every process p_i that is not active in C. Note that if C has no active processes, then the only one round schedule applicable to C is the empty schedule σ, where $\sigma[i] = \emptyset$ for every process p_i.

Suppose that C is a configuration of an algorithm and σ is a one round schedule that is applicable to C. Consider the round in which each process p_i that is active in C sends the messages specified by the algorithm when p_i is in state $C[i]$ to the processes in $\sigma[i]$. Let C' be the configuration resulting from this round when performed from configuration C. Then $C\sigma$ denotes the configuration C'. The processes that *crash during* σ are those that have crashed in C', but have not crashed in C. In other words, p_i crashes during σ if $C'[i] = \perp$ and $C[i] \neq \perp$.

Two configurations C and C' are *indistinguishable* to process p_i when $C[i] = C'[i]$. Suppose that σ and σ' are one round schedules applicable to configurations C and C', respectively. If process p_i receives messages from the same set of processes Q during σ and σ' and configurations C and C' are indistinguishable to p_i and each of the processes in Q, then configurations $C\sigma$ and $C'\sigma'$ are indistinguishable to process p_i.

An *execution* of an algorithm from configuration C is an (infinite or finite) alternating sequence of configurations and one round schedules $C = C_0, \sigma_1, C_1, \sigma_2, C_2, \ldots$ such that σ_r is applicable to C_{r-1} and $C_r = C_{r-1}\sigma_r$ for all rounds $r \geq 1$. The sequence $\sigma = \sigma_1, \sigma_2, \ldots$ is called a *schedule applicable to* C. When σ has length t, it is called a t *round schedule* and $C\sigma$ denotes the configuration C_t. For any set of processes $Q \subseteq P = \{p_0, \ldots, p_{n-1}\}$, a Q*-only schedule* is a schedule in which only processes in Q send messages (i.e., $\sigma[j] \subseteq \{p_j\}$ for every process $p_j \notin Q$ and every round σ in the schedule). Note that, if configurations C and C' are indistinguishable to every process in Q, then the set of Q-only schedules applicable to C and C' are the same.

A *complete execution* ends with a final configuration. The *round complexity* of an algorithm is the maximum number t such that some process is active at the beginning of round t in some execution of the algorithm from an initial configuration. Every algorithm with round complexity t can be converted into an algorithm in which no process outputs a value before the end of round t: A process that outputs a value in round $r < t$ can, instead, simply send no messages in rounds $r+1$ through t and output the value at the end of round t.

3 The FloodMin Algorithm

The FloodMin Algorithm [7, 12] solves k-set agreement among $n \leq k(t+1)$ processes in t rounds, when at most k processes can crash each round. We present this algorithm because it motivates parts of the construction in Section 5.

In the first round of this algorithm, each process broadcasts its input value and then adopts the smallest value from among its own value and the values it received from other processes. In each successive round, each process broadcasts its adopted value and then adopts the smallest value from among its own value and the values it received from other processes. At the end of round t, each process outputs its adopted value.

Let $\alpha = C_0, \sigma_1, C_1, \sigma_2, \ldots, \sigma_t, C_t$ be an execution of FloodMin starting from an initial configuration, let $adopt(C_0)$ denote the set of input values of processes in configuration C_0, and, for each $r \in \{1, \ldots, t\}$, let $adopt(C_r)$ denote the set of values adopted by processes (that have not crashed) in configuration C_r. The proof that at most k different values are output in α is an easy consequence of the following two facts.

▶ **Lemma 1.** *$adopt(C_r) \subseteq adopt(C_{r-1})$ for all $r \in \{1, \ldots, t\}$.*

▶ **Lemma 2.** *If at most $d - 1$ processes crash during round r, then $\#adopt(C_r) \leq d$.*

Since the set of output values, $adopt(C_t)$, is a subset of the set of input values, $adopt(C_0)$, FloodMin satisfies validity. It also satisfies k-agreement for $n \leq k(t+1)$ processes: If there is a round r of α in which fewer than k processes crash, then, by Lemma 2, $\#adopt(C_r) \leq k$ and, hence, by Lemma 1, $\#adopt(C_t) \leq k$. Otherwise, in every round of α, at least k processes crash. Hence, at the end of round t, there are at most $n - kt \leq k$ active processes and, thus, $\#adopt(C_t) \leq k$.

However, when $n = k(t + 1) + 1$, there are executions of FloodMin in which the set of adopted values has size $k + 1$ at the end of round t. For example, consider an initial configuration C'_0 in which $kt + 1$ processes have input k and one process has input i, for each $i \in \{0, \ldots, k - 1\}$. Inductively, for $1 \leq r \leq t$, there is a configuration C'_r in which $k(t - r) + 1$ processes have adopted value k and one process has adopted value i, for each $i \in \{0, \ldots, k - 1\}$. It can be obtained from configuration C'_{r-1} by a one round schedule in which the processes with value k don't crash and the processes with other values each sends one message to a different process with value k and then crashes. In configuration C'_t, there are $k + 1$ processes, each with a different adopted value. If they terminated at the end of round t, then k-agreement would be violated.

4 Extension-Based Proofs

As in the definition of an extension-based proof in the iterated immediate snapshot or iterated snapshot model [2, 3], an extension-based proof for the synchronous message passing model is an interaction between a prover and an algorithm, which the prover is trying to prove is incorrect. The prover starts with no knowledge about the algorithm except for its initial configurations.

Suppose that the algorithm claims to solve a task $\mathcal{T}$ for n processes within t rounds when up to f processes can crash each round. Without loss of generality, we may assume that no process outputs a value before the end of round t. Let M be the set of possible output values for $\mathcal{T}$. There are $t + 1$ phases in the interaction.

In phase 0, the prover asks about the values that can be output in executions from initial configurations. A *query* (C, Q, u, m) consists of an initial configuration C, a set of at least $n - u$ processes Q, an upper bound $u \in \{0, \ldots, f\}$, and a value $m \in M$. The algorithm either:

- responds positively with a Q-only t round schedule σ from C and a process $p_i \in Q$ such that
  - at most u processes crash in each round of σ and
  - p_i has output m in configuration $C\sigma$
- or responds NONE, if there are no such schedule and process.

After a finite number of queries, the prover ends phase 0 by choosing an initial configuration C_0. Then the players proceed to phase 1.

Let $1 \leq r \leq t-1$. At the end of phase $r-1$, the prover has chosen a configuration C_{r-1} that is reachable from an initial configuration by an $r-1$ round schedule in which at most f processes crash each round. During phase r, the prover considers configurations that can be reached from C_{r-1} by one round schedules. It asks about the values that can be output in $t-r$ round executions from such configurations. A *query* $(C_{r-1}, \sigma'_r, u, m)$ consists of an upper bound $u \in \{0, \ldots, f\}$, a value $m \in M$, and a one round schedule σ'_r applicable to C_{r-1} in which at most f processes crash. The algorithm either:

- responds positively with a $t-r$ round schedule σ from $C_{r-1}\sigma'_r$ and a process p_i such that
  - at most u processes crash in any round of σ and
  - p_i has output m in configuration $C_{r-1}\sigma'_r\sigma$
- or responds NONE, if there are no such schedule and process.

After a finite number of queries, the prover ends phase r by choosing a one round schedule σ_r applicable to C_{r-1} in which at most f processes crash and defines $C_r = C_{r-1}\sigma_r$. Then the players proceed to phase $r+1$.

In phase t, the prover asks what values are output by processes at the end of one round executions from configuration C_{t-1}. A *query* $(C_{t-1}, \sigma'_t, p_i)$ consists of a one round schedule σ'_t applicable to C_{t-1}, in which at most f processes crash, and a process p_i, which is not crashed in configuration $C_{t-1}\sigma'_t$. The algorithm responds with the value output by p_i in configuration $C_{t-1}\sigma'_t$.

At the end of phase t, the prover chooses a one round schedule σ_t applicable to C_{t-1} in which at most f processes crash and defines $C_t = C_{t-1}\sigma_t$.

Finally the winner of the interaction is determined. If the outputs in configuration C_t violate the specifications of task $\mathscr{T}$, then the prover wins. If there is contradictory information from the algorithm, then the prover also wins. Otherwise, the prover loses.

An Extension-Based Proof of the Lower Bound for Binary Consensus

Binary consensus is another name for k-set agreement when $k = 1$. As an example, we present an extension-based proof that any algorithm solving binary consensus among $n \geq t+2$ processes requires at least $t+1$ rounds when at most one process crashes each round. It is based on the valency arguments used to prove this result [1, 5, 13].

Suppose that $\mathcal{A}$ is t round algorithm that claims to solve consensus among $n \geq t+2$ processes. We construct a prover that wins against $\mathcal{A}$. Note that, if $\mathcal{A}$ responds negatively to both $(C, Q, u, 0)$ and $(C, Q, u, 1)$ for some initial configuration C, bound $u \in \{0, 1\}$, and set Q of at least $n-u$ processes, then it has provided contradictory information. Specifically, in the t round execution from C in which no processes crash, every process must output 0 or 1. Likewise, if $\mathcal{A}$ responds negatively to both $(C_{r-1}, \sigma'_r, u, 0)$ and $(C_{r-1}, \sigma'_r, u, 1)$, where σ'_r is a one round schedule applicable to C_{r-1}, then it has provided contradictory information. Since providing contradictory information causes $\mathcal{A}$ to lose, we'll suppose that $\mathcal{A}$ never does this.

In phase 0, the prover will try to find a bivalent initial configuration. The prover asks the queries $(D_j, P, 1, 0)$ and $(D_j, P, 1, 1)$, for all $0 \leq j \leq n$, where D_j is the initial configuration in which the first j processes, $p_0, \ldots, p_{j-1}$, have input 0 and the rest have input 1. If there exists j such that $\mathcal{A}$ responds positively to both queries, the prover chooses $C_0 = D_j$ and proceeds to phase 1.

Otherwise, for each j, there exists $m_j \in \{0,1\}$ such that $\mathcal{A}$ responded positively to $(D_j, P, 1, m_j)$ and responded negatively to $(D_j, P, 1, 1-m_j)$. If $m_0 = 0$, then the algorithm violates validity, since all processes in configuration D_0 have input 1. Similarly, if $m_n = 1$, the algorithm violates validity, since all processes in configuration D_n have input 0. So, assume that $m_0 = 1$ and $m_n = 0$. Then there exists $0 \leq j < n$ such that $m_j = 1$ and $m_{j+1} = 0$. Next, the prover asks the queries $(D_j, Q, 1, 0)$ and $(D_j, Q, 1, 1)$, where $Q = P - \{p_j\}$. Since $Q \subseteq P$ and $\mathcal{A}$ responded negatively to $(D_j, P, 1, 0)$, $\mathcal{A}$ must respond negatively to $(D_j, Q, 1, 0)$ and positively to $(D_j, Q, 1, 1)$ to avoid providing contradictory information. Let σ and $p_i \in Q$ be its response to $(D_j, Q, 1, 1)$. Since D_j and D_{j+1} are indistinguishable to every process in Q and σ is a Q-only schedule, p_i also outputs 1 in $D_{j+1}\sigma$. However, this contradicts the negative response $\mathcal{A}$ gave to the query $(D_{j+1}, P, 1, 1)$. Thus, the prover wins.

In phase r, for $1 \leq r \leq t-1$, the prover tries to find a bivalent configuration reachable from C_{r-1} by a one round schedule. The prover asks the queries $(C_{r-1}, \sigma'_r, 1, 0)$ and $(C_{r-1}, \sigma'_r, 1, 1)$ for every one round schedule σ'_r applicable to C_{r-1} in which at most 1 process crashes. If there exists σ'_r such that $\mathcal{A}$ responds positively to both, then $C_{r-1}\sigma'_r$ is bivalent. In this case, the prover chooses $C_r = C_{r-1}\sigma'_r$ and proceeds to phase $r+1$.

Otherwise, for each σ'_r, there exists $m(\sigma'_r) \in \{0,1\}$ such that $\mathcal{A}$ responded positively to $(C_{r-1}, \sigma'_r, 1, m(\sigma'_r))$ and responded negatively to $(C_{r-1}, \sigma'_r, 1, 1-m(\sigma'_r))$. Let α denote the one round schedule applicable to C_{r-1} in which no processes crash. Consider the set of all one round schedules σ'_r applicable to C_{r-1} in which one process crashes and $m(\sigma'_r) \neq m(\alpha)$. This set is nonempty since C_{r-1} is bivalent. Among all schedules in this set, let β be one in which the process p_i that crashes sends the largest number of messages to the processes that are active in C_{r-1}. Let Q be the set of processes that are active in $C_{r-1}\beta$. If p_i sends a message to every process in Q during β (i.e., $Q \subseteq \beta[i]$), then $\mathcal{A}$ has provided contradictory information. Specifically, $C_{r-1}\alpha$ and $C_{r-1}\beta$ are indistinguishable to every process in Q, but $m(\alpha) \neq m(\beta)$. So, suppose there is some process $p_\ell \in Q$ to which p_i does not send a message in β. Let β' be the one round schedule applicable to C_{r-1} that is the same as β except that p_i also sends a message to p_ℓ. Then, by definition of β, it follows that $m(\beta') \neq m(\beta)$. In this case, $\mathcal{A}$ has provided contradictory information, since $C_{r-1}\beta'$ and $C_{r-1}\beta$ are indistinguishable to all processes in $Q - \{p_\ell\}$. Thus, the prover wins.

In phase t, the prover asks the queries $(C_{t-1}, \sigma'_t, p_i)$ for all one round schedules σ'_t applicable to C_{t-1} in which at most 1 process crashes and all processes p_i which have not crashed in $C_{t-1}\sigma'_t$. If $\mathcal{A}$ responded with both 0 and 1 as output values in $C_{t-1}\sigma'_t$, for some one round schedule σ'_t, then the prover wins, since $\mathcal{A}$ has violated 1-agreement.

Otherwise, for each σ'_t, there exists $m(\sigma'_t) \in \{0,1\}$ such that $\mathcal{A}$ responded with $m(\sigma'_t)$ to the queries $(C_{t-1}, \sigma'_t, p_i)$ for all processes p_i which have not crashed. Let α denote the one round schedule applicable to C_{t-1} in which no processes crash. Consider the set of all one round schedules σ'_t applicable to C_{t-1} in which one process crashes and $m(\sigma'_t) \neq m(\alpha)$. This set is nonempty since C_{t-1} is bivalent. Among all schedules in this set, let β be one in which the process p_i that crashes sends the largest number of messages to the processes that are active in C_{t-1}. Let Q be the set of processes excluding p_i that are active in C_{t-1}. If p_i sends a message to every process in Q during β (i.e., $Q \subseteq \beta[i]$), then $\mathcal{A}$ has provided contradictory information. Specifically, $C_{t-1}\alpha$ and $C_{t-1}\beta$ are indistinguishable to every process in Q, but $m(\alpha) \neq m(\beta)$. So, suppose there is some process $p_\ell \in Q$ to which p_i does not send a message in β. Let β' be the one round schedule applicable to C_{r-1} that is the same as β except that p_i also sends a message to p_ℓ. Then, by definition of β, it follows that $m(\beta') \neq m(\beta)$. In this case, $\mathcal{A}$ has provided contradictory information, since $C_{t-1}\beta'$ and $C_{t-1}\beta$ are indistinguishable to all processes in $Q - \{p_\ell\}$. Thus, the prover wins.

5 Why Extension-Based Lower Bounds Fail

This section is devoted to proving the main result of the paper.

▶ **Theorem 3.** *For $k, t \geq 2$, there is no extension-based proof of a lower bound of $t+1$ rounds for solving k-set agreement among $n = k(t+1)+1$ processes when each process has an input in $\{0, \ldots, k\}$ and at most k processes can crash each round.*

To prove this result, we construct a t round adversarial algorithm $\mathcal{A}$ that is able to win against every extension-based prover.

We begin by presenting some terminology and notation used to describe $\mathcal{A}$. For any configuration C and any process p_i that is not crashed in C, let $adopt(C, p_i)$ be the smallest input value process p_i saw in the execution leading to configuration C. If C is an initial configuration, then $adopt(C, p_i) = x_i$, the input value of process p_i. If $C' = C\sigma$, where σ is a one round schedule, and p_i is not crashed in C', then $\{adopt(C, p_j) \mid p_i \in \sigma[j]\}$ is the *set of adopted values that p_i saw during σ*, either because the value was adopted by p_i in C or the value was adopted by some other process that sent a message to p_i during σ. As in FloodMin, $adopt(C', p_i) = \min\{adopt(C, p_j) \mid p_i \in \sigma[j]\}$. For any configuration C, any set of processes Q that are active in C, and any possible value $m \in M = \{0, \ldots, k\}$, let

$$adopt(C, Q) = \{adopt(C, q) \mid q \in Q\}$$

be the *set of values adopted in C by processes in Q*.

The following one round schedules will be useful for defining our adversarial algorithm. Let C be any configuration, let Q be any subset of the active processes in C, and let $m \in M$ be any possible output value.

- In $\alpha(C, Q)$, processes in Q do not crash and all other active processes crash without sending any messages:

$$\alpha(C,Q)[i] = \begin{cases} \{p_0, \ldots, p_n\} & \text{if } p_i \in Q \\ \{p_i\} & \text{if } p_i \text{ is active in } C, \text{ but } p_i \notin Q \\ \phi & \text{if } p_i \text{ is not active in } C. \end{cases}$$

- In $\beta(C, Q, m)$, processes in Q that have adopted a value other than m in C do not crash and all other active processes crash without sending any messages:

$$\beta(C,Q,m)[i] = \begin{cases} \{p_0, \ldots, p_n\} & \text{if } p_i \in Q \text{ and } adopt(C,p_i) \neq m \\ \{p_i\} & \text{if } p_i \text{ is active in } C, \text{ but } p_i \notin Q \text{ or } adopt(C,p_i) = m \\ \phi & \text{if } p_i \text{ is not active in } C. \end{cases}$$

If $m \notin adopt(C, Q)$, then $\beta(C, Q, m)$ is the same as $\alpha(C, Q)$. Otherwise, it is like $\alpha(C, Q)$, except that the processes in Q that have adopted m in C crash before sending any messages.

- In $\beta(C, Q, {<}m)$, processes in Q that have adopted a value greater than or equal to m in C do not crash and all other active processes crash without sending any messages:

$$\beta(C,Q,{<}m)[i] = \begin{cases} \{p_0, \ldots, p_n\} & \text{if } p_i \in Q \text{ and } adopt(C,p_i) \geq m \\ \{p_i\} & \text{if } p_i \text{ is active in } C, \text{ but } p_i \notin Q \text{ or } adopt(C,p_i) < m \\ \phi & \text{if } p_i \text{ is not active in } C. \end{cases}$$

If $adopt(C, Q) \subseteq \{m, m+1, \ldots, k\}$, then $\beta(C, Q, m)$ is the same as $\alpha(C, Q)$. Otherwise, it is like $\alpha(C, Q)$, except that the processes in Q that have adopted values less than m in C crash before sending any messages.

If C is a configuration at the end of round $r < t-1$ and γ is a one round schedule starting from C, then γ^* is the $t-r$ round schedule where the first round is γ and no processes crash in the remaining $t-r-1$ rounds.

Algorithm $\mathcal{B}(m^*)$

Before defining the adversarial algorithm $\mathcal{A}$, we consider $k+1$ different bad algorithms, $\mathcal{B}(m^*)$, one for each value $m^* \in M$. Except for what they output, processes in $\mathcal{B}(m^*)$ behave as in FloodMin, with each active process repeatedly trying to send its adopted value to all other processes and adopting the smallest value it saw. At the end of round t, every process p_i that has not crashed will output a value (but not necessarily the smallest value) that it saw in the last round. Specifically, let a_i be the value that p_i adopted at the end of round $t-1$. Then p_i outputs a_i, except in two special cases.

- During round t, if p_i received at least $k+1$ messages, no message with value a_i, and at least one message with every other value, then it outputs the smallest value it saw at least twice.
- If $a_i = m^*$ and p_i saw each value in M exactly once during round t, then it outputs $(m^*+1) \bmod (k+1)$.
- Otherwise, p_i outputs a_i.

Note that, if p_i saw each value in M exactly once, then it received exactly k messages, so these two cases are mutually exclusive.

Since the set of adopted values at each round is a subset of the input values, every output value is an input value. Hence $\mathcal{B}(m^*)$ satisfies validity.

Consider any configuration C'_{t-1} reachable from an initial configuration by a $t-1$ round schedule in which at most k processes crash each round, any one round schedule σ'_t applicable to C'_{t-1} in which at most k processes crash, and any process p_i that has not crashed in $C'_t = C'_{t-1}\sigma'_t$. The following three observations are consequences of the definition of $\mathcal{B}(m^*)$. They are true because neither of the special cases are applicable.

▶ **Observation 4.** *In algorithm $\mathcal{B}(m^*)$, if process p_i saw at most k different values during σ'_t, then p_i outputs adopt(C'_{t-1}, p_i) in configuration C'_t.*

▶ **Observation 5.** *In algorithm $\mathcal{B}(m^*)$, if process p_i saw the value it adopted in C'_{t-1} at least twice during σ'_t, then p_i outputs adopt(C'_{t-1}, p_i) in configuration C'_t.*

▶ **Observation 6.** *Let $m \in M$. In algorithm $\mathcal{B}(m^*)$, if process p_i did not see m during round σ'_t, then p_i does not output m in configuration C'_t.*

Here is another useful property of this algorithm.

▶ **Lemma 7.** *Let $m \in M$. In algorithm $\mathcal{B}(m^*)$, if process p_i saw m at most once, saw every other value in M at least once, and received at least $k+1$ messages during σ'_t, then p_i does not output m in configuration C'_t.*

Proof. If p_i did not see m during σ'_t, then, by Observation 6, it does not output m in C'_t. So, suppose that p_i saw m exactly once during σ'_t. Also suppose that, during σ'_t, p_i saw every other value in M at least once and it received at least $k+1$ messages.

If p_i saw the value it adopted in C'_{t-1} at least twice during σ'_t, then, by Observation 5, it outputs *adopt*$(C'_{t-1}, \sigma'_t) \neq m$ in configuration C'_t. Therefore, suppose that p_i saw the value it adopted in C'_{t-1} only once during σ'_t. Then, during σ'_t, it received no message with this value and received messages with every other value. Since the conditions of the first special case are satisfied, p_i outputs the smallest value it saw at least twice, which is not m. ◀

If at most k different values were adopted at the end of round $t-1$, it doesn't matter which of these values a process outputs. So suppose that each of the $k+1$ values in M was adopted by some process at the end of round $t-1$. By Lemmas 1 and 2, this can only occur if k processes crashed in each of the first $t-1$ rounds. Hence, there are $2k+1$ active processes at the end of round $t-1$. At least one value in M was adopted by at least two active processes. Furthermore, at most k values in M were adopted by at least two active processes. Thus, a process can output a value that it sees at least twice during round t. It is also possible that k processes crash at the beginning of round t and each of the remaining $k+1$ processes sees each of the $k+1$ values in M exactly once during round t. In $\mathcal{B}(m^*)$, this symmetry is broken by having the process that had adopted m^* at the end of round $t-1$ output a different value. If m^* is a value that was adopted by at most one process at the end of round $t-1$, then these two approaches don't interfere with one another. However, for each $m^* \in M$, there is some final configuration of algorithm $\mathcal{B}(m^*)$ in which all $k+1$ values in M are output. We explicitly construct such a configuration in the appendix.

The Adversarial Algorithm

During phases 0 through $t-1$, the adversarial algorithm $\mathcal{A}$ responds to each query in a way that is consistent with all these bad algorithms. At the end of phase $t-1$, the adversarial algorithm chooses one of these bad algorithms and responds as if it is that algorithm during phase t. Its choice depends on the prover's choice of configuration C_{t-1}. The chosen bad algorithm has the property that it does not violate *k-agreement* in any final configuration reachable from C_{t-1}. Next, we give detailed specifications for how the adversarial algorithm responds to queries in each phase. Then we show that at most k different values are output in the configuration chosen by the prover at the end of phase t. Finally, to show that $\mathcal{A}$ never answers queries inconsistently, we prove that $\mathcal{A}$ responded to every query in a way that is consistent with the bad algorithm it chose.

Phase 0

Consider any query (C, Q, u, m) made by the prover during phase 0, where C is an initial configuration, Q is a set of processes, $u \in \{0, \ldots, k\}$ is an upper bound on the number of crashes per round, and $m \in M$ is a value.

- If m is the input of some process in Q (i.e., $m \in adopt(C, Q)$) and there are at most u processes that are not in Q or have inputs less than m, then $\mathcal{A}$ responds with the schedule $\beta^*(C, Q, {<}m)$ (in which these processes crash immediately) and the process $p_i \in Q$ with smallest index that has input m in configuration C.
- Otherwise, $\mathcal{A}$ responds with NONE.

The following observations are useful consequences of this specification.

▶ **Observation 8.** *If $\mathcal{A}$ responds to the query (C, Q, u, m) in phase 0 with $\beta^*(C, Q, {<}m)$ and p_i, then m is the only value p_i saw in the last round of $\beta^*(C, Q, {<}m)$.*

▶ **Observation 9.** *If $\mathcal{A}$ responds to the query (C, Q, u, m) in phase 0 with NONE, then, for every configuration C'_{t-1} reachable from C by a $(t-1)$-round Q-only schedule in which at most u processes crash each round, $m \notin adopt(C'_{t-1}, Q'_{t-1})$, where Q'_{t-1} is the set of active processes in C'_{t-1}.*

Phases $1, \ldots, t-2$

In phase r, where $1 \leq r \leq t-2$, the adversarial algorithm $\mathcal{A}$ behaves similarly. Consider any query $(C_{r-1}, \sigma'_r, u, m)$ made by the prover during phase r, where σ'_r is a one round schedule applicable to C_{r-1} in which at most k processes crash, $u \in \{0, \ldots, k\}$ is an upper bound on the number of crashes per round, and $m \in M$ is a value. Let $C'_r = C_{r-1}\sigma'_r$ and let Q'_r be the set of active processes in C'_r.

- If m is the adopted value of some process in Q'_r (i.e., $m \in \mathit{adopt}(C'_r, Q'_r)$) and there are at most u processes in Q'_r with adopted value less than m, then $\mathcal{A}$ responds with the schedule $\beta^*(C'_r, Q'_r, {<}m)$ (in which these processes crash immediately) and the process $p_i \in Q'_r$ with smallest index that has adopted m in C'_r.
- Otherwise, $\mathcal{A}$ responds with NONE.

The following two observations are analogous to Observation 8 and Observation 9.

▶ **Observation 10.** *If $\mathcal{A}$ responds to the query $(C_{r-1}, \sigma'_r, u, m)$ in phase $1 \leq r \leq t-2$ with $\beta^*(C'_r, Q'_r, {<}m)$ and p_i, where Q'_r is the set of active processes in $C'_r = C'_{r-1}\sigma'_r$, then m is the only value p_i saw in the last round of $\beta^*(C'_r, Q'_r, {<}m)$.*

▶ **Observation 11.** *If $\mathcal{A}$ responds to the query $(C_{r-1}, \sigma'_r, u, m)$ in phase $1 \leq r \leq t-2$ with* NONE*, then, for every configuration C'_{t-1} reachable from $C_{r-1}\sigma'_r$ by a $(t-1-r)$-round schedule in which at most u processes crash each round, $m \notin \mathit{adopt}(C'_{t-1}, Q'_{t-1})$, where Q'_{t-1} is the set of active processes in C'_{t-1}.*

Note that, during phases 0 through $t-2$, the adversarial algorithm responds in a way that is consistent with FloodMin.

Phase $t-1$

In phase $t-1$, the adversarial algorithm's strategy is different. It depends on the query and the prover's choice for C_{t-2}. Consider any query $(C_{t-2}, \sigma'_{t-1}, u, m)$ made by the prover during phase $t-1$, where σ'_{t-1} is a one round schedule applicable to C_{t-2} in which at most k processes crash, $u \in \{0, \ldots, k\}$ is an upper bound on the number of crashes per round, and $m \in M$ is a value. Let $C'_{t-1} = C_{t-2}\sigma'_{t-1}$ and let Q'_{t-1} be the set of active processes in C'_{t-1}. The response chosen by $\mathcal{A}$ depends on the values adopted by these processes in configuration C'_{t-1}.

- If $m \notin \mathit{adopt}(C'_{t-1}, Q'_{t-1})$, then $\mathcal{A}$ responds NONE.

So, suppose $m \in \mathit{adopt}(C'_{t-1}, Q'_{t-1})$. Let $i = \min\{j \mid p_j \in Q'_{t-1} \text{ and } \mathit{adopt}(C'_{t-1}, p_j) = m\}$ be the smallest index of a process that adopted m. Let $m' \in M - \{m\}$ be the smallest value other than m that was adopted in C'_{t-1} by the fewest number of processes. In particular, it is possible that $m' \notin \mathit{adopt}(C'_{t-1}, Q'_{t-1})$.

- If at most u processes adopted m', then $\mathcal{A}$ responds with the one round schedule $\beta(C'_{t-1}, Q'_{t-1}, m')$ (in which these processes crash immediately) and process p_i.
- If more than u processes adopted m' and at least 2 processes adopted m, then $\mathcal{A}$ responds with the one round failure-free schedule $\alpha(C'_{t-1}, Q'_{t-1})$ and process p_i.
- Otherwise, $\mathcal{A}$ responds NONE.

The information communicated by positive and negative responses in phase $t-1$ is different than in the previous phases.

▶ **Lemma 12.** *If $\mathcal{A}$ responds to the query $(C_{t-2}, \sigma'_{t-1}, u, m)$ in phase $t-1$ with σ and p_i, then p_i had adopted value m in $C_{t-2}\sigma'_{t-1}$ and either p_i saw at most k different values during σ or p_i saw m at least twice during σ.*

Proof. Suppose that $\mathcal{A}$ responds to $(C_{t-2}, \sigma'_{t-1}, u, m)$ in phase $t-1$ with σ and p_i. By construction, $adopt(C_{t-2}\sigma'_{t-1}, p_i) = m$. Let m' be the smallest value other than m that was adopted in $C'_{t-1} = C_{t-2}\sigma'_{t-1}$ by the fewest number of processes in Q'_{t-1}.

If at most u processes adopted m' in C'_{t-1}, none of these processes sent any messages in $\sigma = \beta(C'_{t-1}, Q'_{t-1}, m')$. Then, during σ, p_i did not see m', so it saw at most k different values.

Otherwise, more than u processes adopted m' in C'_{t-1}, at least 2 processes adopted m in C'_{t-1}, and $\sigma = \alpha(C'_{t-1}, Q'_{t-1})$. By definition, no processes crash in σ, so p_i saw m at least twice during σ. ◀

▶ **Lemma 13.** *If $\mathcal{A}$ responds to the query $(C_{t-2}, \sigma'_{t-1}, u, m)$ in phase $t-1$ with* NONE, *then*

- *$m \notin adopt(C'_{t-1}, Q'_{t-1})$ or*
- *exactly one process in Q'_{t-1} adopted m in C'_{t-1} and*
  more than u, but at most 2, processes in Q'_{t-1} adopted m' in C'_{t-1},

where Q'_{t-1} is the set of active processes in C'_{t-1} and $m' \in M - \{m\}$ is the smallest value other than m that was adopted by the fewest number of processes in C'_{t-1}.

Proof. Suppose that $\mathcal{A}$ responds to $(C_{t-2}, \sigma'_{t-1}, u, m)$ in phase $t-1$ with NONE and $m \in adopt(C'_{t-1}, Q'_{t-1})$. Then, from the specifications of $\mathcal{A}$, more than u processes in Q'_{t-1} adopted m' and less than 2 processes in Q'_{t-1} adopted m in configuration C'_{t-1}. Since $m \in adopt(C'_{t-1}, Q'_{t-1})$, exactly one process in Q'_{t-1} adopted m in C'_{t-1}.

By definition of m', each value $m'' \in M - \{m\}$ was adopted in C'_{t-1} by at least as many processes in Q'_{t-1} as m' was. Since m' was adopted by more than $u \geq 0$ processes in Q'_{t-1}, it follows that $m'' \in adopt(C'_{t-1}, Q'_{t-1})$. Hence $\#adopt(C'_{t-1}, Q'_{t-1}) = k+1$. By Lemma 2, at least k processes crashed in each of the $t-1$ rounds of the execution from C_0 to C'_{t-1}. At most k processes can crash each round, so exactly k processes crashed in each of these rounds and $\#Q'_{t-1} = n - k(t-1) = 2k+1$. Since only one process in Q'_{t-1} has adopted m, the other $2k$ processes in Q'_{t-1} have each adopted one of the k values in $M - \{m\}$. Thus, m', which was adopted by the fewest number of these processes, was adopted by at most 2 of these processes. ◀

Phase t

Let Q_{t-1} be the set of active processes in C_{t-1} and let $m^* \in M$ be the smallest value that was adopted in C_{t-1} by the fewest number of processes in Q_{t-1}. In particular, if $\#adopt(C_{t-1}, Q_{t-1}) \leq k$, then m^* was adopted by no process in Q_{t-1}. However, if $\#adopt(C_{t-1}, Q_{t-1}) = k+1$, then, by Lemma 2, at least k processes crashed in each of the $t-1$ rounds of the execution from C_0 to C_{t-1}. At most k processes can crash each round, so exactly k processes crashed in each of these rounds and $\#Q_{t-1} = n - k(t-1) = 2k+1$. In this case, the number of processes in Q_{t-1} that adopted m^* is at most $\lfloor \#Q_{t-1}/(k+1) \rfloor = \lfloor (2k+1)/(k+1) \rfloor = 1$.

Consider any query $(C_{t-1}, \sigma'_t, p_i)$ made by the prover during phase t, where σ'_t is a one round schedule applicable to C_{t-1} in which at most k processes crash and p_i is a process that has not crashed in $C_{t-1}\sigma'_t$. Then $\mathcal{A}$ responds with $adopt(C_{t-1}, p_i)$, except in two special cases.

- If p_i received at least $k+1$ messages, no message with value $adopt(C_{t-1}, p_i)$, and at least one message with every other value, then $\mathcal{A}$ responds with the smallest value p_i saw at least twice during σ'_t.
- If $adopt(C_{t-1}, p_i) = m^*$ and p_i saw each value in M exactly once during σ'_t then $\mathcal{A}$ responds with $(m^*+1) \bmod (k+1)$.

- Otherwise, $\mathcal{A}$ responds with $\mathit{adopt}(C_{t-1}, p_i)$.

Note that, during σ'_t, if p_i saw each value in M exactly once, then it received exactly k messages, so the two special cases are mutually exclusive.

When $\mathcal{A}$ responds to a query with value m^*, the execution has special properties.

▶ **Lemma 14.** *If $\mathcal{A}$ responds to the query $(C_{t-1}, \sigma'_t, p_i)$ in phase t with m^*, then $\mathit{adopt}(C_{t-1}, p_i) = m^*$ and p_i saw at most k different values during σ'_t.*

Proof. Since at most one process in Q_{t-1} adopted m^* in C_{t-1}, process p_i saw m^* at most once during σ'_t, so $\mathcal{A}$ does not respond with m^* as a result of the first special case. Since $m^* \neq (m^* + 1) \bmod k$, $\mathcal{A}$ does not respond with m^* as a result of the second special case. Thus, $\mathit{adopt}(C_{t-1}, p_i) = m^*$.

To obtain a contradiction, suppose that p_i saw all $k + 1$ values during σ'_t. Since the second special case does not hold, there is a value $m \in M$ that p_i saw at least twice during σ'_t. Hence, p_i received at least $k + 1$ messages during σ'_t. Since p_i is the only process in Q_{t-1} that adopted m^* in C_{t-1}, it did not receive a message with value $m^* = \mathit{adopt}(C_{t-1}, p_i)$, but it did receive at least one message with every other value. But then the first special case holds, which is a contradiction. Hence, p_i saw at most k different values during σ'_t. ◀

The following observation is a consequence of the specifications, since neither of the special cases is applicable.

▶ **Observation 15.** *If p_i saw the value it adopted in C_{t-1} at least twice during σ'_t, then $\mathcal{A}$ responds to the query $(C_{t-1}, \sigma'_t, p_i)$ in phase t with $\mathit{adopt}(C_{t-1}, p_i)$.*

Agreement

Let C_t be the the final configuration chosen by the prover at the end of phase t and let σ_t be the one round schedule such that $C_t = C_{t-1}\sigma_t$. We show that $\mathcal{A}$ does not lose by violating k-agreement in C_t.

▶ **Lemma 16.** *Consider the set of processes Q_t that output values in C_t. $\mathcal{A}$ responds to the queries (C_{t-1}, σ_t, p_i) for $p_i \in Q_t$ with at most k different values.*

Proof. Suppose not. For each $m \in M$, let $q_m \in Q_t$ be the process with smallest index that outputs m. By Lemma 14, process q_{m^*} saw at most k different values during σ_t. During σ_t, process q_{m^*} received a message from every process in Q_t, so $\#\mathit{adopt}(C_{t-1}, Q_t)$ is bounded above by the number of different values q_{m^*} saw during σ_t. Thus $\mathit{adopt}(C_{t-1}, Q_t) \subsetneq M$. Consider the sequence $m_0, m_1, \ldots, m_{k+1}$, where

- $m_0 = \min(M - \mathit{adopt}(C_{t-1}, Q_t))$ is the smallest value not in $\mathit{adopt}(C_{t-1}, Q_t)$ and
- $m_i = \mathit{adopt}(C_{t-1}, q_{m_{i-1}}) \in M$ is the value adopted by $q_{m_{i-1}}$ in C_{t-1}, for all $i \geq 1$.

Since $\#M = k + 1$, the sequence contains at least one duplicate. Let j be the smallest positive integer such that $m_j = m_i$ for some $i < j$.

Note that $m_i \neq m_0$, since $m_0 \notin \mathit{adopt}(C_{t-1}, Q_t)$, but $m_j = \mathit{adopt}(C_{t-1}, q_{m_{j-1}}) \in \mathit{adopt}(C_{t-1}, Q_t)$. By the minimality of j, $m_{j-1} \neq m_{i-1}$, so $q_{m_{j-1}}$ and $q_{m_{i-1}}$ output different values in C_t, but they adopted the same value $m_j = m_i$ in C_{t-1}. During σ_t, processes $q_{m_{i-1}}$ and $q_{m_{j-1}}$ receive a message with this value from one another. Therefore both $q_{m_{j-1}}$ and $q_{m_{i-1}}$ saw m_j at least twice during σ_t. By Observation 15, both these processes output the values they had adopted in C_{t-1}. Hence $m_{j-1} = \mathit{adopt}(C_{t-1}, q_{m_{j-1}}) = m_j = m_i = \mathit{adopt}(C_{t-1}, q_{m_{i-1}}) = m_{i-1}$, which is a contradiction. ◀

Consistency

To show that the responses of the adversarial algorithm $\mathcal{A}$ to the queries do not contradict one another, we show that they are all consistent with algorithm $\mathcal{B}(m^*)$, where m^* is the smallest value adopted by the fewest number of processes in C_{t-1}, the configuration chosen by the prover at the end of phase $t-1$. Note that the choice of m^* depends on the choices made by the prover.

First, we show that, whenever $\mathcal{A}$ responded positively to a query, $\mathcal{B}(m^*)$ can give the same response. The queries made in each phase are considered separately.

▶ **Lemma 17.** *If $\mathcal{A}$ responded to a query with a schedule and a process (in phases 0 to $t-1$) or with a value (in phase t), then that response is consistent with algorithm $\mathcal{B}(m^*)$.*

Proof. Suppose the prover asked query (C, Q, u, m) in phase 0 and $\mathcal{A}$ responded with $\beta^*(C, Q, {<}m)$ and p_i. Let σ'_t be the last round of this schedule and let C'_{t-1} be the second last configuration in the execution of $\beta^*(C, Q, {<}m)$ from C. Then $C\beta^*(C, Q, {<}m) = C'_{t-1}\sigma'_t$. By Observation 8, m was the only value p_i saw in σ'_t. In particular, $adopt(C'_{t-1}, p_i) = m$. Then, by Observation 4, in algorithm $\mathcal{B}(m^*)$, process p_i outputs $adopt(C'_{t-1}, p_i) = m$ in configuration $C'_{t-1}\sigma'_t$.

Suppose the prover asked query $(C_{r-1}, \sigma'_r, u, m)$ in phase r, where $1 \leq r \leq t-2$, and $\mathcal{A}$ responded with $\beta^*(C'_r, Q'_r, {<}m)$ and p_i. Then $C'_r = C_{r-1}\sigma'_r$ and Q'_r is the set of active processes in C'_r. Let σ'_t be the last round of the schedule $\beta^*(C'_r, Q'_r, {<}m)$ and let C'_{t-1} be the second last configuration in the execution of $\beta^*(C'_r, Q'_r, {<}m)$ from C'_r. By Observation 10, m is the only value p_i saw in σ'_t. This implies that $adopt(C'_{t-1}, p_i) = m$ and, hence, by Observation 4, in algorithm $\mathcal{B}(m^*)$, p_i outputs m in configuration $C'_{t-1}\sigma'_t$.

Suppose the prover asked query $(C_{t-2}, \sigma'_{t-1}, u, m)$ in phase $t-1$. Let $C'_{t-1} = C_{t-2}\sigma'_{t-1}$, let Q'_{t-1} be the set of active processes in C'_{t-1}, and let $m' \in M - \{m\}$ be the smallest value other than m that was adopted by the fewest number of processes in C'_{t-1}.

If $\mathcal{A}$ responded with $\beta(C'_{t-1}, Q'_{t-1}, m')$ and p_i, then at most u processes in Q'_{t-1} adopted m' in configuration C'_{t-1} and $adopt(C'_{t-1}, p_i) = m$. Since all processes in Q'_{t-1} that adopted value m' crash without sending any messages in $\beta(C'_{t-1}, Q'_{t-1}, m')$, it follows that, during $\beta(C'_{t-1}, Q'_{t-1}, m')$, process p_i did not see m' and, hence, saw at most k different values. By Observation 4, in algorithm $\mathcal{B}(m^*)$, process p_i outputs $adopt(C'_{t-1}, p_i) = m$ in configuration $C'_{t-1}\sigma'_t$.

If $\mathcal{A}$ responded with $\alpha(C'_{t-1}, Q'_{t-1})$ and p_i, then at least two processes adopted m and $adopt(C'_{t-1}, p_i) = m$. Since no processes crash in $\alpha(C'_{t-1}, Q'_{t-1})$, process p_i received a message from every other process in Q'_{t-1}, so p_i saw m at least twice during $\alpha(C'_{t-1}, Q'_{t-1})$. By Observation 5, in algorithm $\mathcal{B}(m^*)$, process p_i outputs $adopt(C'_{t-1}, p_i) = m$ in configuration $C'_{t-1}\sigma'_t$.

Finally, suppose the prover asked query $(C_{t-1}, \sigma'_t, p_i)$ in phase t and $\mathcal{A}$ responded with m. Since the same three cases also occur in the specification of $\mathcal{B}(m^*)$, p_i outputs m in configuration $C_{t-1}\sigma'_t$ of algorithm $\mathcal{B}(m^*)$. ◀

Next, we show that whenever $\mathcal{A}$ responded negatively to a query, $\mathcal{B}(m^*)$ also responds negatively. Again, we consider the queries made in each phase separately.

▶ **Lemma 18.** *If $\mathcal{A}$ responded to a query with* NONE, *then algorithm $\mathcal{B}(m^*)$ responds to the query with* NONE.

Proof. Suppose the prover asked query (C, Q, u, m) in phase 0 and $\mathcal{A}$ responded with NONE. Then, by Observation 9, for every configuration C'_{t-1} reachable from C by a $(t-1)$-round Q-only schedule in which at most u processes crash each round, $m \notin \mathit{adopt}(C'_{t-1}, Q'_{t-1})$, where Q'_{t-1} is the set of active processes in C'_{t-1}. Therefore, for every one round schedule σ'_t applicable to C'_{t-1} in which at most u processes crash, every process that is not crashed in $C'_{t-1}\sigma'_t$ did not see m during σ'_t. By Observation 6, in algorithm $\mathcal{B}(m^*)$, p_i does not output m in configuration $C'_{t-1}\sigma'_t$. Hence, $\mathcal{B}(m^*)$ also responds with NONE to the query (C, Q, u, m) in phase 0.

Suppose the prover asked query $(C_{r-1}, \sigma'_r, u, m)$ in phase r, where $1 \le r \le t-2$, and $\mathcal{A}$ responded with NONE. Then, by Observation 11, for every configuration C'_{t-1} reachable from $C_{r-1}\sigma'_r$ by a $(t-1-r)$-round schedule in which at most u processes crash each round, $m \notin \mathit{adopt}(C'_{t-1}, Q'_{t-1})$, where Q'_{t-1} is the set of active processes in C'_{t-1}. Therefore, for every one round schedule σ'_t applicable to C'_{t-1} in which at most u processes crash, every process that is not crashed in $C'_{t-1}\sigma'_t$ did not see m during σ'_t. By Observation 6, in algorithm $\mathcal{B}(m^*)$, p_i does not output m in configuration $C'_{t-1}\sigma'_t$. Hence, $\mathcal{B}(m^*)$ also responds with NONE to the query $(C_{r-1}, \sigma'_r, u, m)$ in phase r.

Suppose the prover asked query $(C_{t-2}, \sigma'_{t-1}, u, m)$ in phase $t-1$ and $\mathcal{A}$ responded with NONE. Let $C'_{t-1} = C_{t-2}\sigma'_{t-1}$ and let Q'_{t-1} be the set of active processes in C'_{t-1}. If $m \notin \mathit{adopt}(C'_{t-1}, Q'_{t-1})$, then, for every one round schedule σ'_t applicable to C'_{t-1} in which at most u processes crash, every process that is not crashed in $C'_{t-1}\sigma'_t$ did not see m during σ'_t. By Observation 6, in algorithm $\mathcal{B}(m^*)$, p_i does not output m in configuration $C'_{t-1}\sigma'_t$. Hence $\mathcal{B}(m^*)$ also responds with NONE to the query $(C_{t-1}, \sigma'_t, u, m)$ in phase $t-1$.

Therefore, suppose that $m \in \mathit{adopt}(C'_{t-1}, Q'_{t-1})$. By Lemma 13, exactly one process in Q'_{t-1} adopted m in configuration C'_{t-1} and more than u, but at most 2, processes in Q'_{t-1} adopted m' in configuration C'_{t-1}, where $m' \in M - \{m\}$ is the smallest value other than m that was adopted by the fewest number of processes in C'_{t-1}. Let σ'_t be an arbitrary one round schedule applicable to C'_{t-1} in which at most u processes crash and let p_i be a process that is not crashed in $C'_{t-1}\sigma'_t$. Since only one process in Q'_{t-1} adopted m in configuration C'_{t-1}, process p_i saw m at most once during σ'_t.

By definition of m', each value $m'' \in M - \{m\}$ was adopted in C'_{t-1} by at least as many processes in Q'_{t-1} as m' was. Since m' was adopted by more than $u \ge 0$ processes in Q'_{t-1} and at most u processes crash in σ'_t, process p_i saw m'' during σ'_t.

At most k processes crash in each of the first $t-1$ rounds and at most $u < 2$ processes crash in σ'_t, so p_i receives at least $n-1-k(t-1)-1 = 2k-1$ messages. Note that $2k-1 \ge k+1$ since $k \ge 2$. Hence, by Lemma 7, in algorithm $\mathcal{B}(m^*)$, process p_i does not output m in configuration $C'_{t-1}\sigma'_t$. Therefore, $\mathcal{B}(m^*)$ also responds with NONE to the query $(C_{t-1}, \sigma'_t, u, m)$ in phase $t-1$. ◀

6 Conclusions

In this paper, we define the class of extension-based proofs for synchronous message passing models and study the power of such proofs. On one hand, we give an an extension-based proof of the t round lower bound for solving binary consensus among $n \ge t+1$ processes when at most one process can crash each round. On the other hand, we show that, for $k \ge 2$ and $t > 2$, there is no extension-based proof of the t round lower bound for solving k-set agreement among $n = kt+1$ processes when at most k processes can crash each round.

There are a number of problems that remain open. First, is there an extension-based proof of the t round lower bound for solving k-set agreement among $n > kt+1$ processes if at most k processes can crash each round, for $k \ge 2$? If so, what is the smallest value of n for which such a proof exists? A related problem is whether there is an extension-based proof of the t round lower bound for solving k-set agreement among among $n = kt + 1$ processes when any number of processes can crash in each round.

There is a simple 1 round lower bound for solving k-set agreement among $n \geq k + 1$ processes. Without any communication, every process must output its input value to ensure validity. Hence, in any initial configuration in which all $k + 1$ values in M appear as inputs, either validity or k-agreement is violated.

Is there an extension-based proof of the 2 round lower bound for solving k-set agreement among $n = 2k + 1$ processes? The proof of Theorem 3 does not work in this case, since Observation 8 and Observation 9 do not always hold. The reason it may be difficult to extend the result to include this case is that there are more queries the prover can ask in phase $t - 1$ when $t = 1$ than when $t > 1$. In phase 0, for any bound u, value m, and initial configuration C'_0, the prover can specify a set Q of at most $n - u$ processes and ask whether there is a one round Q-only schedule in which at most u processes crash and some process outputs m when applied to configuration C'_0. However, when $t > 1$, for any bound u, value m, and configuration $C'_{t-1} = C_{t-2}\sigma'_{t-1}$, in phase $t - 1$, the prover can only ask whether there is a one round schedule in which at most u processes crash and some process outputs m when applied to configuration C'_{t-1}. In particular, for $u > 0$, the prover cannot specify a subset Q of processes that must crash at the beginning of the one round schedule. This restriction gives the adversary more flexibility when choosing what to answer, which we take advantage of in our proof.

Our choice of allowable queries in the definition of extension-based proofs was influenced by the valency argument showing the lower bound for consensus. Specifically, to make extension-based proofs interesting in the synchronous message passing model, this valency argument should be able to be expressed as an extension-based proof. Although it suffices to only use queries in which there is no additional restriction on the number of processes that can crash each round (i.e., $u = k$), also allowing queries with smaller values of u makes the prover stronger and, hence, makes Theorem 3 better. Other definitions for extension-based proofs are certainly possible. It would be interesting to see if a more restricted class of queries allows Theorem 3 to be extended to $n > k(t + 1) + 1$ or $t = 1$, or makes its proof significantly easier.

Finally, we would like to show that there are other problems for which extension-based proofs cannot be used to obtain known lower bounds on the number of rounds necessary to solve them.

References

1 Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that t-resilient consensus requires $t + 1$ rounds. *Inf. Process. Lett.*, 71(3-4):155–158, 1999.

2 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In *Proceedings of the 51st Annual ACM Symposium on Theory of Computing (STOC)*, pages 986–996, 2019.

3 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Brief announcement: Why extension-based proofs fail. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 54–56, 2020.

4 Hagit Attiya, Armando Castañeda, and Sergio Rajsbaum. Locally solvable tasks and the limitations of valency arguments. In *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS)*, volume 184 of *LIPIcs*, pages 18:1–18:16, 2020.

5 Hagit Attiya and Faith Ellen. *Impossibility Results for Distributed Computing*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2014.

6 Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 1993.

7 Soma Chaudhuri, Maurice Herlihy, Nancy A. Lynch, and Mark R. Tuttle. Tight bounds for k-set agreement. *J. ACM*, 47(5):912–943, 2000.

8 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 143–152, 1998.

9 Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. An overview of synchronous message-passing and topology. *Electron. Notes Theor. Comput. Sci.*, 39(2):1–17, 2000.

10 Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. An axiomatic approach to computing the connectivity of synchronous and asynchronous systems. *Electron. Notes Theor. Comput. Sci.*, 230:79–102, 2009.

11 Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

12 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

13 Yoram Moses and Sergio Rajsbaum. A layered analysis of consensus. *SIAM J. Comput.*, 31(4):989–1021, 2002.

14 Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

A Why Algorithm $\mathcal{B}(m^*)$ Violates k-Agreement

To show that algorithm $\mathcal{B}(m^*)$ violates *k-agreement*, we construct a final configuration C'_t of $\mathcal{B}(m^*)$ in which each of the $k+1$ values in M is output by some process. As in the lower bound for FloodMin, presented at the end of Section 3, we start from an initial configuration in which there are $k+1$ different input values and construct an execution in which k processes crash each round and the set of adopted values has size $k+1$ after each of the first $t-1$ rounds.

Let C'_0 be an initial configuration in which process p_i has input $m \in \{0,\ldots,k-1\}$ for $i = k(t+1)-m$ and has input k for $0 \leq i \leq kt$. In other words, the first $n-k = kt+1$ process have input k and the last k processes have inputs $k-1$ to 0. Inductively, we construct a configuration C'_r at round r, for $1 \leq r \leq t-2$, in which process p_i

- adopted $m \in \{0,\ldots,k-1\}$ for $i = k(t+1-r)-m$
- adopted k for $0 \leq i \leq k(t-r)$, and
- is crashed for $k(t+1-r) < i \leq n-1$.

In other words, the first $n-k(r+1) = k(t-r)+1$ process adopted k, the next k processes adopted $k-1$ to 0, and the last kr processes are crashed.

For $1 \leq r \leq t-2$, consider the one round schedule σ'_r applicable to C'_{r-1} in which

$$\sigma'_r[i] = \begin{cases} \{p_i, p_{i-k}\} & \text{for } k(t+1-r) < i \leq k(t+2-r), \\ P & \text{for } 0 \leq i \leq k(t+1-r), \text{ and,} \\ \phi & \text{for } k(t+2-r) < i < n. \end{cases}$$

In other words, the first $n-kr = k(t-r-1)+1$ processes do not crash and the next k processes each crash after sending a single message, to the process whose index is k less than its own index. Let $C'_r = C'_{r-1}\sigma'_r$. Note that, for $0 \leq i \leq k(t-r)$, the only messages process p_i received during σ'_r were from the first $k(t-r-1)+1$ processes, all of which had adopted value k in C'_{r-1}, so p_i keeps k as its adopted value in C'_r. However, for $k(t-r) < i \leq k(t-r+1)$, process p_i also received a message from process p_{i+k}, which had adopted value $m = k(t+1-(r-1))-(i+k) = k(t+1-r)-i$ in C'_{r-1}, so p_i adopts value m in C'_r.

Consider the one round schedule σ'_{t-1} applicable to C'_{t-2} in which

$$\sigma'_{t-1}[i] = \begin{cases} \{p_i, p_{i-k}, p_0\} & \text{if } m^* \neq k \text{ and } i = 3k - m^*, \\ \{p_i, p_{i-k}\} & \text{for } 2k < i < 3k - m^* \text{ and } 3k - m^* < i \leq 3k, \\ P & \text{for } 0 \leq i \leq 2k, \text{ and}, \\ \phi & \text{for } 3k < i \leq n-1. \end{cases}$$

In other words, the first $n - k(t-1) = 2k+1$ processes do not crash, and the next k processes crash after sending one or two messages. These k processes each send a message to the process whose index is k less than its own index. In addition, if $m^* \neq k$, process p_{3k-m^*} also sends a message to process p_0. Let $C'_{t-1} = C'_{t-2}\sigma'_{t-1}$. Note that, for $0 < i \leq k$, the only messages process p_i received during σ'_{t-1} were from the first $2k+1$ processes, all of which had adopted value k in C'_{t-2}, so p_i keeps k as its adopted value in C'_{t-1}. For $k < i \leq 2k$, process p_i also received a message from process p_{i+k}, which had adopted value $m = k(t+1-(r-1)) - (i+k) = k(t+1-r) - i$ in C'_{t-2}, so p_i adopts value m in C'_{t-1}. If $m^* = k$, process p_0 adopts value m^*. If $m^* \neq k$, process p_0 also received a message from p_{3k-m^*}, which had adopted value m^* in C'_{t-2}, so p_0 also adopts value m^*.

Finally, consider the one round schedule σ'_t in which

$$\sigma'_t[i] = \begin{cases} \{p_0, p_{2k-m^*}\} & \text{for } i = 0, \\ \{p_i\} & \text{for } 1 \leq i \leq k-1, \\ P & \text{for } k \leq i \leq 2k, \text{ and} \\ \phi & \text{for } 2k+1 \leq i < n-1. \end{cases}$$

Let $C'_t = C'_{t-1}\sigma'_t$. Note that, for $k \leq i \leq 2k$, process p_i sent the value $2k - i$ it adopted in C'_{t-1} to every other process during σ'_t. Thus, each process p_i saw each value in M at least once during σ'_t. If $i \neq 2k - m^*$, then p_i received no other messages during σ'_t, so it outputs $2k - i \neq m^*$ by the specifications of $\mathcal{B}(m^*)$. If $i = 2k - m^*$, then process p_i also received a message with value m^* from process p_0 during σ'_t, so it outputs $2k - i = m^*$ by the specifications of $\mathcal{B}(m^*)$. Hence, $k+1$ different values are output in configuration C'_t.

Truthful Information Dissemination in General Asynchronous Networks

Lior Solodkin ✉
Blavatnik School of Computer Science, Tel Aviv University, Israel

Rotem Oshman ✉
Blavatnik School of Computer Science, Tel Aviv University, Israel

Abstract

We give a protocol for information dissemination in asynchronous networks of rational players, where each player may have its own desires and preferences as to the outcome of the protocol, and players may deviate from the protocol if doing so achieves their goals. We show that under minimalistic assumptions, it is possible to solve the information dissemination problem in a *truthful* manner, such that no participant has an incentive to deviate from the protocol we design. Our protocol works in any asynchronous network, provided the network graph is at least 2-connected. We complement the protocol with two impossibility results, showing that 2-connectivity is necessary, and also that our protocol achieves optimal bit complexity.

As an application, we show that truthful information dissemination can be used to implement a certain class of *communication equilibria*, which are equilibria that are typically reached by interacting with a trusted third party. Recent work has shown that communication equilibria can be implemented in synchronous networks, or in asynchronous, complete networks; we show that in some useful cases, our protocol yields a lightweight mechanism for implementing communication equilibria in any 2-connected asynchronous network.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Algorithmic game theory and mechanism design

Keywords and phrases game theory, asynchronous networks, information dissemination

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.37

Funding *Rotem Oshman*: Research funded by the Israel Science Foundation, Grant No. 2801/20.

1 Introduction

Consider a network of n rational players trying to jointly carry out some distributed task, such as computing a spanning tree or electing a leader. Each player has its own desires and preferences: e.g., some players may want to be close to the root of the tree, while others may want to minimize their degree, to reduce traffic flowing through them. The players are given a protocol that can solve the task, but since they are self-interested, they may *deviate* from the protocol if doing so would serve their interests. Can we design a distributed protocol that solves the problem, such that faithfully following the protocol is an *equilibrium* – that is, no player has an *incentive* to deviate from the protocol? This question is at the heart of the recent line of work on *game theory in distributed computing* [2, 3, 1, 5, 6, 15, 4, 23]; we now know that it is possible, for example, to choose a leader on the ring uniformly at random, regardless of the players' individual preferences, and even to do it in a manner resilient to *coalitions* between the players (e.g., [4, 23, 15]).

In this paper we focus on one building block which is used in many distributed protocols: *information dissemination*. We assume that each player i has some piece of information x_i, initially known only to i, and we would like all players to learn all the information of all the players. Information dissemination is typically solved by *flooding* or by *pipelining* [16]. However, these protocols do not work well when the players are self-interested: what is to prevent a player i from lying about its input, or lying about *other players'* inputs, which i is supposed to forward? In this work we study information dissemination with rational players in general asynchronous networks, and give a simple communication-optimal protocol, under some minimalistic assumptions.

Solving the information dissemination problem does require some assumptions about the *uilities* of the players – their desires and preferences. For example, if player 1's input is $x_1 \in \{0, 1\}$, and player 1's utmost desire is to have all players output that $x_1 = 0$, then whenever $x_1 = 1$, player 1 will lie about its input throughout the protocol; nothing we do can prevent that. Truthful information dissemination is only possible when players have no a-priori reason to lie about their inputs, in the absence of any information about the other players' inputs. However, as soon as anything is learned about the other inputs, a player may have an incentive to lie about its own input, so as to influence the final outcome. For example, mechanisms for auctions often assume *sealed bids*, and may no longer remain truthful if players have information about the other players' bids when placing their own bid.

In addition to the assumption above ("no a-prior reason to lie"), we also need to assume that players want the protocol to succeed, or at least not to abort; a similar assumption, called *solution preference*, is made in prior work (e.g., [4, 23, 5, 14, 6, 15]).

Our protocol shows that these two assumptions together – that player have no a-priori incentive to lie, and that they do not want the protocol to obviously fail – are sufficient to solve information dissemination in a truthful manner in any 2-connected communication network.

Application to communication equilibria. Part of the motivation for our work is to help extend the scope of rational distributed algorithms, beyond algorithms that merely choose a *uniformly random* solution from a set of legal solutions (as in, e.g., [4, 23, 5, 6]). A uniformly random solution is *fair*, but it may not be *good*. For example, instead of electing a uniformly random leader, as was done in [4, 23], we might wish to take the players' preferences into account, and elect the leader that maximizes the players' happiness in some sense. This motivates us to search for efficient distributed implementations of game-theoretic mechanisms for finding a "good" solution, rather than a uniformly random solution.

The particular mechanism we consider here is *communication equilibria* [12], a type of equilibrium that is usually reached with the help of a *mediator*. In classical game theory, a mediator is a trusted entity that helps the players reach an equilibrium: the players communicate with the mediator, and the mediator then suggests a course of action to each player. Even though the players are free to lie to the mediator or ignore its recommendation, it is known that adding a mediator can yield better payoffs for the players – it can enable us to reach an equilibrium in games where the same equilibrium could not be reached without a mediator [12, 8, 9]. For example, mediators are useful in *congestion games* [22], which can be used to model routing of packets in a network (see, e.g., [21]).

In [2, 1] and others it is shown that in some scenarios, a distributed network of rational players can *simulate* a mediator even when we do not have access to a real one; this enables us to implement the rich class of communication equilibria in the distributed setting. This type of simulation is referred to in the game theory community as "cheap talk" – the players can communicate freely before deciding what actions to take. However, so far, all existing literature on implementing a mediator in distributed networks has either assumed that the network is *synchronous*, or that it is *fully-connected*. In addition, previous solutions are heavily based on tools from cryptography – e.g., cryptographic primitives such as *envelope* and *ballot-box* in [17, 19], or *secure multi-party computation* (secure MPC) in [2, 1]. These tools enable the players to simulate the mediator by securely computing the mediator's output, in such a way that no player finds out the private information of any other player. However, this privacy comes at a cost: the cryptographic primitives used in the literature are fairly heavyweight to implement, and require larger messages and higher running time compared to plain computation where the inputs are not kept private. As a result, in addition to the limitation of having a network that is either synchronous or fully-connected, these protocols also incur a high overhead.

As an application of our algorithm, we identify several natural scenarios where privacy of the inputs is not required to achieve an equilibrium, and information dissemination can be used to implement a mediator, by simply having the players collect all inputs and compute the mediator's recommendations. In these scenarios, our information-dissemination algorithm yields a lightweight, communication-optimal mediator implementation in general, 2-connected asynchronous networks. We prove that our algorithm is indeed communication-optimal, by proving a lower bound on the number of bits required to implement a mediator in a general network (even in synchronous networks).

1.1 Our Results

A truthful information-dissemination protocol. In Section 4, we construct a randomized protocol that solves information dissemination in any 2-connected network, and has the property that no player can improve their expected payoff by deviating from the protocol (that is, following the protocol is an *equilibrium*). Our protocol uses $O(n)$ messages, $O(n^2 \cdot b)$ total communication bits, and $O(D)$ asynchronous rounds, where D is the diameter of the network graph and b is the number of bits required to represent a single input.

Following [10, 13], we do not consider coalitions of players, or Byzantine attacks; these are interesting questions to consider in future work. We conjecture that our protocol extends to coalitions of $k > 1$ players, assuming the network is at least $(k+1)$-connected. We also assume that the network graph is known in advance. This assumption is common to most of the work on game-theoretic distributed computing, with some exceptions, e.g., [5, 6]; these, too, often focus on particular *classes* of graphs, such as rings.

Application to computing communication equilibria. We refer to the type of equilibria that can be computed by our protocol as *full-information communication equilibria*. In Section 6, we give two natural scenarios where a given communication equilibrium is also a full-information equilibrium: the first is any game where a player's payoff depends only on its own output (and not the outputs of the other players), and the second is any distributed task where

(a) the players prefer to reach a valid solution, and

(b) in any valid solution, given the outputs of any set of $n-1$ players, there is a unique valid output for the last player.

The second scenario includes *agreement* problems, such as leader election, but also other useful distributed primitives.

Lower bounds. One might wonder whether it is *necessary* to collect all players' inputs in order to achieve a communication equilibrium. To complement our results, we prove that the answer is "yes": there is some game with a full-information communication equilibrium that requires $\Omega(n)$ messages and $\Omega(n^2 \cdot b)$ total bits of communication to achieve, where b is the number of bits required to represent the players' types and actions. This means that using our information-dissemination protocol to compute full-information communication equilibria is optimal, in the worst-case. We also prove that the assumption of 2-connectivity, which is made by our protocol, is necessary: in any network that is not 2-connected, there is some game with a full-information communication equilibrium that cannot be achieved.

A remark about our solution concept. In the game theory community, many flavors of equilibria have been considered. Following [4], the solution concept we adopt in this paper is *sequential equilibrium* [18]. Our protocol, like that of [4], involves "collective punishment": if a player detects that another player has deviated from the protocol, it aborts the protocol, thereby punishing *itself* as well as all the other players. One may view this as a "non-credible threat" – why should any rational player actually carry out such an action? Following [4], we showed that our protocol achieves a *sequential equilibrium* [18], a solution concept formulated to handle non-credible threats. However, for the sake of accessibility, in the current paper we focus on the weaker but more familiar concept of a *Nash equilibrium*.

2 Related Work

Information dissemination. Information dissemination is a central building block in distributed algorithms, and a goal in itself. In fault-free sychronous networks, a communication and time-efficient solution is *pipelining* [16], and the problem has also been studied in a variety of other network models.

To our knowledge, the only work to consider information dissemination in the game-theoretic setting is [5], where the problem is referred to as *knowledge sharing.* Several protocols for knowledge sharing are given in [5]: for synchronous rings, for synchronous or asynchronous complete networks, and for synchronous general networks. General asynchronous networks are not considered in [5]. Moreover, [5] makes a strong assumption, called *full knowledge*, about the protocol that *uses* knowledge sharing as a building block: "for each agent that does not know the values of **all other agents**, any output of the protocol is still equally possible" [5] (emphasis ours). In particular, this implicitly assumes that the protocol that uses the knowledge-sharing building block chooses a uniformly random solution. In contrast, here we assume only that agents have no incentive to lie about their input in the absence of *any* information about the other players' inputs; as soon as a player has learned even a single bit of another player's input, it may have an incentive to lie about its own input.

Implementing a mediator. Our paper studies distributed implementations of a mediator by cheap-talk protocols. This topic was first investigated by the economics community; the most directly relevant work to ours is [10], which solves the same problem in a similar setting, but it requires a fully-connected, synchronous network, and has high message complexity, $\Theta(n^2)$. [13] gave a protocol that implements rational secret sharing and secure multi-party computation (secure MPC), and these results were extended in [2] to protocols that withstand a coalition of rational players, or some players that are Byzantine instead of rational. A mediator was implemented in [2] using secure MPC, but [2] required again a fully-connected synchronous network. In [3], lower bounds were given for the size of a coalition we can withstand when implementing a mediator.

The first work to implement a mediator in an asynchronous network was [1], which showed that in a fully-connected asynchronous network, we can implement a mediator and even handle coalitions and non-rational players. Since the protocols of [1] again rely on secure MPC, their message complexity is $O(n \cdot N \cdot c)$, where n is the number of players, N is the number of messages in the mediated game, and c is a bound on the number of arithmetic gates required to implement the mediator as an arithmetic circuit. In this paper, by considering the restricted class of *full-information* communication equilibria, we obtain an improved message complexity of $O(n)$, and handle general asynchronous networks.

We remark that some of the works mentioned above achieve different solution concepts than the type of equilibrium we aim for in this paper (sequential equilibrium). For example, [13, 2] both obtain a Nash equilibrium that is resilient to the iterated deletion of weakly-dominated strategies. This type of equilibrium, unlike sequential equilibrium, is not concerned with what action the player should take in states that occur off the equilibrium path.

Other related work. Several recent works considered fair solutions to specific distributed problems: electing a uniformly random leader was studied in [4, 23], and other problems and building blocks were considered in [5, 14, 6, 15]. The solution concept we suggest in this paper can solve some of these problems more generally: for example, choosing a uniformly random leader is one type of full-information communication equilibrium, which ignores the utilities of the players, but there are other equilibria that take the utilities into account (for example, we can maximize the *social welfare*, the sum of the players' utilities). On the negative side, our protocol is not resilient against coalitions, while several of the aforementioned works can handle coalitions and even Byzantine players. Also, like much of the prior work, we do assume that the network graph is known to all the players in advance. (This is also assumed in the secure MPC implementation of [20].)

3 Preliminaries

Games and protocols. In this paper we are concerned with two types of games: the game we are trying to implement – that is, the problem specification – is a *normal-form Bayesian game*, where players choose one action, and then their utility is determined by the actions that all the players have taken. Our implementation, a protocol for asynchronous networks, is formally modeled as an *extensive-form Bayesian game*, which represents all possible executions in the asynchronous network, and captures the interactive nature of such executions. For the sake of brevity, we define only normal-form Bayesian games here, as extensive-form games are not necessary to understand our protocol. (We touch more on extensive-form games in Section 5, where we give a high-level overview of the correctness proof for our protocol.)

Notation. Given an n-tuple $s = (s_0, \dots, s_{n-1})$, we let $s_{-i} = (s_0, \dots, s_{i-1}, s_{i+1}, \dots, s_{n-1})$ denote the $(n-1)$-tuple obtained by omitting s_i. When i is clear from the context, the notation (s_i, s_{-i}) represents the original n-tuple s.

Normal-form Bayesian games. A normal-form (or strategic-form) Bayesian game is defined as follows. Note that the *types* referred to in the definition correspond to *inputs* in our case.

▶ **Definition 1.** *A* normal-form Bayesian game, $\Gamma = (N, (T_i)_{i \in N}, p, (A_i)_{i \in N}, (u_i)_{i \in N})$, *consists of:*

1. *A set of players N. We usually assume for simplicity that $N = [n]$, where $n \geq 2$ is the number of players.*
2. *For each player $i \in N$, a set of possible types (inputs) T_i. Let $T = T_0 \times \dots T_{n-1}$.*
3. *A probability distribution $p : T \to [0, 1]$ over the types of the players.*
4. *For each player $i \in N$, a set of possible actions A_i. Let $A = A_0 \times \cdots \times A_{n-1}$.*
5. *For each player $i \in N$, a utility function $u_i : T \times A \to \mathbb{R}$.*

The game is played as follows: first, the types are drawn from their distribution, $t = (t_0, \dots, t_{n-1}) \sim p$. Each player i is given only its own type t_i, but the prior p is known to all players. Next, each player i independently chooses an action $a_i \in A_i$, and the utility of each player is then determined by $u_i((t_0, \dots, t_{n-1}), (a_0, \dots, a_{n-1}))$.

Strategies and equilibria. A *mixed strategy* for player i in the game Γ is a function mapping player i's type $t_i \in T_i$ to a probability distribution over player i's actions A_i; a *strategy profile* is an n-tuple $s = (s_0, \dots, s_{n-1})$ assigning to each player $i \in N$ a mixed strategy s_i. Player i's *expected utility* for the game Γ under the strategy profile $s = (s_0, \dots, s_{n-1})$ is given by $\mathbb{E}_{t \sim p|t_i, a \sim s(t)} [u_i(t, a)]$. Here, t_i denotes player i's type, and $s(t)$ denotes the distribution on n-tuples of actions where the i-th element is sampled independently from $s_i(t_i)$.

A strategy s_i for player i is called *best response* to $s_{-i} = (s_0, \dots, s_{i-1}, s_{i+1}, s_{n-1})$ if it maximizes player i's expected utility, assuming that the other players play according to s_{-i}.

A strategy profile $s = (s_0, \dots, s_{n-1})$ is called a (mixed) *Nash equilibrium* if for each player i, the strategy s_i is player i's best response to s_{-i}.

Mediators. Some normal-form games may have strategy profiles that are desirable for some reason – for example, they might lead to high social welfare – but which are not Nash equilibria. In such cases, it can be helpful to enlist the help of an external *mediator*. Intuitively, a mediator is a trusted entity; the players tell their types to the mediator, and the mediator then provides each player with a recommended action. The players are free to lie to the mediator or ignore its recommendation, but a good mediator will render such deviations not profitable (in expectation).

Formally, a mediator d is a mapping from n-tuples of types to distributions on n-tuples of actions, and we would like it to satisfy the following:

▶ **Definition 2.** *A mediator d is a* communication equilibrium *of normal-form Bayesian game Γ if for all $i \in N, t_i, t_i' \in T_i, \alpha_i : A_i \to A_i$:*

$$\sum_{t_{-i} \in T_{-i}} p(t_{-i}|t_i) \sum_{a \in A} d(a|t) u_i(t, a) \geq \sum_{t_{-i} \in T_{-i}} p(t_{-i}|t_i) \sum_{a \in A} d(a|t_{-i}, t_i') u_i(t, (\alpha_i(a_i), a_{-i})).$$

Here, $d(a|t)$ represents the probability that the mediator returns $a \in A$ when the players send it $t \in T$, and $d(a|t_{-i}, t_i')$ represents the probability that the mediator returns $a \in A$ when player i sends t_i', and the other players send t_{-i}.

Intuitively, the definition asserts that for each type t_i, there is no "lie" t_i' that player i could tell the mediator, or different action $\alpha(a_i)$ that player i could take instead of the recommendation a_i of the mediator, that would increase player i's expected payoff.

Full-information equilibria. We introduce a subclass of communication equilibria, characterized by being resilient to the revelation of the other players' types and recommended actions:

▶ **Definition 3** (Full-information communication equilibrium)**.** *A mediator d is a* full-information communication equilibrium *for a game $\Gamma = (N, (T_i)_{i \in N}, p, (\tau_i)_{i \in N}, (A_i)_{i \in N}, (u_i)_{i \in N})$ if for each player $i \in [n]$ and type $t_i \in T_i$,*

1. *The player has no incentive to lie about its type: for each $t_i' \in T_i$, and $a_i' \in A_i$,*

$$\sum_{t_{-i} \in T_i} p(t_{-i}|t_i) \sum_{a \in A} d(a|t) u_i(t, a) \geq \sum_{t_{-i} \in T_i} p(t_{-i}|t_i) \sum_{a \in A} d(a|t_{-i}, t_i') u_i(t, (a_i', a_{-i})).$$

2. *After receiving the mediator's recommendation and learning all the other players' types, the player is still not incentivized to deviate: for each* $t_{-i} \in T_{-i}, a \in \sup(d(\cdot|t_i, t_{-i}))$, $\alpha_i : T \times A_{-i} \to A_i$,

$$u_i((t_i, t_{-i}), a) \geq u_i((t_i, t_{-i}), (\alpha_i((t_i, t_{-i}), a), a_{-i})).$$

Here, $\sup(d(\cdot|t)) \subseteq A$ *denotes the support of the mediator's distribution* d *when the players send it the types* t.

The difference from the standard definition of communication equilibria is in the second condition, where we reveal to player i the types and actions of the other players, not just its own recommended action.

Punishment actions. When implementing a general communication equilibrium, we need to assume that players have some means to retaliate against players that deviate from the protocol. As is standard (see, e.g., [10]), we assume that the game contains a *punishment action*, and any outcome where some player carries out this action yields the worst possible payoff for all players.

The information dissemination problem. We model information dissemination as a Bayesian game, Γ_{info}, where the *actions* of the players consist of outputting n-tuples of types, or aborting the protocol; that is, if $T_1, \ldots, T_n$ are the possible types (i.e., inputs) of each of the n players (respectively), then the actions of each player i are $A_i = (T_1 \times \ldots \times T_n) \cup \{\mathsf{abort}\}$. The utility function u_i of each player i then takes the true types $t \in T_1 \times \ldots \times T_n$ of the players, and the outputs $t'_1, \ldots, t'_n \in T_1 \times \ldots \times T_n$ of the players, and returns the payoff $u_i(t, (t'_1, \ldots, t'_n))$ of player i.

As we explained in Section 1, a few assumptions about the utilities of the players are necessary. First, we assume that abort is a *punishment action*, as explained above (that is, players prefer to avoid it above all else). The second assumption is that players have no a-priori incentive to lie about their input, and also that, having learned the other players' inputs, they will correctly output what they have learned (instead of outputting something else). The cleanest and most general way to model this assumption is to define a specific mediator, which we call d_{info}, which gives each player the types of all the other players, and require that the utilities be such that d_{info} is a full-information communication equilibrium. This captures the requirement that in the absence of any information about the types of the other players, no player has an incentive to lie about its own type, and that once the types are learned, the player should output them correctly.

▶ **Definition 4** (The information-dissemination mediator, d_{info}.). *Let* $d_{\mathsf{info}} : T_1 \times \ldots \times T_n \to ((T_1 \times \ldots \times T_n) \cup \{\mathsf{abort}\})^n$ *be the mediator for* Γ_{info} *that returns to each player the types of all the other players:* $d_{\mathsf{info}}(t_1, \ldots, t_n) = (t_1, \ldots, t_n)^n$.

We say that Γ_{info} is a *feasible information-dissemination game* if d_{info}, the mediator from Definition 4, is a full-information communication equilibrium for Γ_{info}.

4 The Protocol

We now present the protocol that players are supposed to follow. Our protocol can implement any given full-information communication equilibrium d for a normal-form Bayesian game Γ, and in particular, the protocol solves the information-dissemination problem: if Γ is a feasible information-dissemination game (as defined in Section 3), then we can have the players simulate the mediator d_{info} to output the types of all the other players.

Throughout the protocol, if any player i detects a deviation from the protocol by any other player, player i irrevocably decides to take its punishment action, guaranteeing the worst possible payoff for all players. In particular, player i chooses to punish if it receives a message from player j that *could not have been sent* in accordance with the protocol in any execution (e.g., if player j is not supposed to message player i at this stage of the protocol, or if the message is ill-formatted).[1]

4.1 High-Level Overview

The goal of our protocol is to get all players to truthfully reveal their types to the entire network, and then have each player locally simulate the mediator and output the action the mediator recommends (or, if we simply want to solve information dissemination, output the types that it has learned). We must manage the process of revealing the players' types carefully: for example, if player i learns the type of player j before player i has said anything about its own type, then player i may have an incentive to lie about its type in order to improve its expected utility. (Recall that the definition of a communication equilibrium guarantees that the players have no incentive to lie in the absence of *any* information about the other players' types; once they have even partial information, all bets are off.)

The main idea underlying our protocol is to

(a) use a basic form of secret sharing to have players *commit* to their type, so that they cannot lie about their type in the future; and, at the same time,

(b) ensure that no player i can learn anything about the type of another player $j \neq i$ before player i has committed to its type.

The protocol has four main stages, which we describe here at a high level, omitting many details. A more detailed description will be given next.

Commitment on a cycle. We fix in advance some shortest cycle C in the graph (recall that the graph must be 2-connected). Assume w.l.o.g. that cycle C comprises players $0, 1, \ldots, \ell$, in this order (with $\ell + 1$ being the length of the cycle).

In the first stage of the protocol, players 0 and 1 *commit to their types*, by "splitting" each type into two shares, each of which reveals nothing about the type by itself. Together both shares reveal the type. The shares are sent along opposite sides of the cycle, in a way that ensures that neither player can receive both shares of the other player before it has sent out both of its own shares, thereby committing to its type in a way that will reveal any future attempt to lie about it.

Commitment on a tree. For the next part of the protocol, we fix in advance a spanning tree T rooted at player 0 and excluding player 1. We proceed through the tree top-down, and have each player interact with its children in the tree in a pairwise commitment protocol, where the parent and the child reveal their types to one another. We ensure that a player never reveals its type to its parent in the tree before the parent is committed – when we start this phase, player 0 is already committed, and we maintain this invariant as we proceed

[1] We note that player i cannot always detect the fact that player j sent it a message when j was not supposed to, because player i does not know where player j is in its execution of the protocol, but in some cases it is obvious – e.g., if player j is supposed to wait for a message from player i before sending a message to i, and i has not yet sent that message. We address the "undetectable" cases in our proof that the protocol is an equilibrium.

down the tree. At the same time, we make sure the player does not *know* its parent's type, only that the parent has already committed to the type. Thus, neither parent nor child have an incentive to deviate from the protocol.

Revealing the types. By the end of the previous part of the protocol, each player has revealed its type to at least one other player. We now use the spanning tree to share all the types with all the players, by simply collecting them up the tree and then broadcasting them downwards.

Detecting deviations. The last stage of the protocol checks whether any player has been "two-faced" and claimed that it has different types at different points in the protocol, or whether some player has tried to lie about the type of another player. To do this, we fix a sparse 2-connected subgraph G' of the network graph in advance, and simply have each player send all the types it received in the previous stage to its neighbors in G'. Each player verifies that the types its neighbors have learned match what it has learned itself. Since G' is 2-connected, between every two players $j, k \neq i$ there is a path π that does not contain i; if player i has given "two different versions" of its type to players j, k, this inconsistency will be discovered, as two neighboring nodes along the path π will have received different versions of i's type.

4.2 Detailed Description

We now give a detailed description of the protocol. Let b be the number of bits required to represent the type of a player, and let R be the number of random bits used by the mediator.

Each player i starts the protocol with a *secret*, denoted s_i. For all players except players 0 and 1, the secret is simply the player's type, $s_i = t_i$. Players 0 and 1 play a special role in the protocol, and at the beginning of the protocol, they generate random strings that will be used later in the protocol. These random strings are part of the secrets of players 0 and 1: for each $i \in \{0, 1\}$, the secret s_i of player i is a tuple consisting of

- The type $t_i \in \{0, 1\}^b$ of player i;
- A private random string $r_i \in \{0, 1\}^R$, which will later be used to simulate the mediator; and
- In the case of player 1, an additional private random string $v \in \{0, 1\}^b$, which will be used in the next stage of the protocol.

Each player $i \in [n]$ has a local array called *values*, of length $n + 1$, where player i stores the secrets of the other players as it learns them. Each player i initializes all cells in the range $\{0, \ldots, n - 1\}$ to $\perp$, except for cell i, which player i initializes to $values[i] = s_i$. The last cell, $values[n]$, is initialized to $\perp$ by all players except player 1, who sets $values[n] = v$.

During the algorithm, cells are updated only by calling the subroutine **Store**(i, x), which ensures that once a value is written to the *values* array, any future attempt to overwrite it with a different value will cause the player to execute the punishment action. The notation $values_i[j]$ refers to cell j of the local *values* array of player i.

Step 1: Commitment in a Cycle

Let C be some fixed shortest cycle in the network graph, and assume w.l.o.g. that the nodes of C are consecutively named $0, 1, \ldots, \ell$. In this part of the protocol, players 0 and 1 commit to each other, as follows: first, each player splits its secret s_i into two shares – a uniformly random "key", $k_i \in \{0,1\}^{|s_i|}$, and an "encryption" of its secret, $e_i = s_i \oplus k_i$. The secret s_i can be reconstructed by taking $k_i \oplus e_i$, but each share k_i, e_i by itself is uniformly random and conveys no information about s_i.

The goal now is for players 0,1 to exchange their shares in such a way that neither player receives *both* of the other player's shares before it has sent out both of its own shares. That is, player 0 can receive *either* k_1 or e_1 before sending out both of its shares, but it should not receive both (and vice-versa). To achieve this, we have each player send one of its shares directly to the other player, and the other share is sent along the other side of the cycle. The order in which shares are sent is orchestrated carefully:

1. First, player 0 sends e_0 to player 1.
2. After receiving e_0, player 1 releases both of its shares: it sends k_1 to player 0, and e_1 to player 2.
3. After receiving k_1, player 0 sends k_0 to player ℓ. At the same time, e_1 is forwarded along the cycle from player 2 to player ℓ.
4. Agent ℓ waits until it has received both k_0 and e_1. Only then does it forward the two messages, sending e_1 to player 0 and k_0 to player $\ell - 1$.
5. Finally, k_0 is forwarded along the cycle from player $\ell - 1$ to player 1.

After receiving e_0 and k_0, player 1 stores the secret $s_0 = e_0 \oplus k_0$ in $values_1[0]$, and player 0 stores $s_1 = e_1 \oplus k_1$ in $values_0[1]$.

Step 2: Commitment in a Tree

Let T be a precomputed BFS tree over the network $G \backslash \{1\}$, rooted at player 0 and excluding player 1. Let H be the height of T. In this part of the protocol, we proceed through the breadth-first layers of T in a top-down manner, and at each step, each player in the current layer executes a short type-exchange protocol with its children in the tree, learning their types and revealing its own type to them. The result is that after $0 \leq h \leq H$ steps, each player i at distance at most h from player 0 in T has revealed its type t_i to some other player $j \neq i$ (where j is the parent of i in the tree if $i \neq 0$, and $j = 1$ if $i = 0$). We will later use this commitment to verify that player i has truthfully revealed its type to the entire network.

Recall that at the beginning of the protocol, player 1 chose a random string v, and this string was revealed to player 0 at the end of the cycle-commitment stage (because it is part of player 1's secret). We now use v as a "secret key" that parents use to commit to their type when they interact with their children in the tree. The value of v itself is also propagated down the tree, so that at the end of this stage, all nodes of the network know it; thus, we can later verify that all commitments were honored.

Let $i \neq 0$ be some player in the tree, and let p be the parent of i. The parent-child exchange protocol between players i and p is executed as follows:

1. The parent p commits to its type by sending its child the message $e_p = t_p \oplus v$.
2. The child i responds by sending its type t_i to p. The parent uses **Store** to store this value in $values_p[i]$. Note that at this point, i knows nothing about the types of the other players, so it has no incentive to lie about its type.
3. The parent "unpacks" its commitment by sending its type t_p to the child, and the child stores t_p in $values_i[p]$ and $e_p \oplus t_p$ in $values_i[n]$.

We point out one subtlety of the protocol: there is nothing to stop a player p in the tree from "fishing for information" by prematurely contacting its child i, sending it a garbage message, and eliciting in return the type of player i. However, our protocol safeguards against this behavior, by making sure that player p has no incentive to do so: if p sends a "garbage message" instead of $e_p = t_p \oplus v$, then the child will discover this later on, when both t_p and v are revealed to all players, and it will punish p.

Steps 3 and 4: Revealing the Types and Detecting Deviations

Once the second stage of the protocol completes, every player in the network is committed to its type, and it is now safe to reveal all the types to everyone. This is carried out by first collecting all the secrets up the tree to the root, and then broadcasting all the values down the tree, so that each player in the network learns the secrets of all the other players. As they learn new types (or more generally, secrets), players use the **Store** function to store them in their local *values* array. Player 0, who is the root of the tree, sends $values_0$ to player 1, who is not in the tree. At the end of this step, every player verifies that its *values* array is full: if player i has some cell j such that $values_i[j] = \bot$, then player i takes its punishment action in the underlying game.

In the final stage of the protocol, we disseminate the secrets collected in the previous stages on a sparse 2-connected subgraph G' of G, and then have each agent verify that in the previous stage, it received the same values as its neighbors did.

To construct G', we show that any 2-connected graph on n vertices has a spanning, 2-connected subgraph with $2n$ edges:

▶ **Proposition 5.** *Suppose that G is a 2-connected graph on n vertices. Then there is a spanning 2-connected subgraph G' of G with $2n$ edges.*

The protocol for detecting whether any player has been inconsistent is simple: each player i compares its *values* array with its two neighbors in G', and if any neighbor has a different *values* array (that is, if even one cell is different), player i executes its punishment action. If this step succeeds at all players that follow the protocol, and at most one player deviates, then all these players will use the same *values* array when they simulate the mediator (next).

In addition, player i verifies that, if $values_i[1] = (\tilde{t}_1, \tilde{r}_1, \tilde{v})$ (recall that player 1's secret has three fields), then $values_i[n] = \tilde{v}$. If this step succeeds, it ensures that i's parent p has honored its commitment, that is, it correctly sent $e_p = t_p \oplus v$ when it first communicated with player i.

Taking action. All players know in advance the mediator function d, which takes a vector $t \in \{0,1\}^{n \cdot b}$ of n types and a string $r \in \{0,1\}^R$, and returns an n-tuple of the mediator's recommended actions on types t using r as the mediator's randomness.

Once the protocol is completed, each player i extracts from its *values* array the types of all the players, and the random strings r_0, r_1. It now simulates the mediator by computing $a = d(t, r_0 \oplus r_1)$, and outputting its recommended action a_i.

5 Sketch of the Correctness Proof

We give an informal overview of the proof that our protocol correctly implements the given full-information communication equilibrium.

Protocols as extensive-form games. As we mentioned in Section 3, we model a distributed protocol as an *extensive-form Bayesian game*, which represents all possible executions of the protocol. An extensive-form game is a tree, where each vertex represents a possible state of the system, and the edges of the tree represent actions of players or the environment (the scheduler). An extensive-form game Γ' is called an *extension* of a normal-form game Γ if the types (inputs) in Γ' are the same as in Γ, and each leaf of Γ' is labeled with an action profile $(a_1, \ldots, a_n) \in A_1 \times \ldots \times A_n$ from Γ. In other words, we think of Γ' as "filling in" what happens in Γ between the point where players are assigned their types, and the point where players output their actions.

Informally, we say that a protocol P *implements* a communication equilibrium d for a normal-form game Γ if, in the extensive-form game induced by P and Γ, for every possible schedule,

(a) following the protocol P is a Nash equilibrium (no player has an incentive to deviate), and

(b) the actions output at every leaf are chosen according to d (that is, if the players' types are t, then the actions output are distributed according to $d(\cdot|t)$).

Proving that our protocol implements the desired communication equilibrium. It is not difficult to see that if all players follow the protocol, then the distribution of their actions is exactly the distribution that the mediator would produce; we focus on proving that our protocol is a Nash-equilibrium, that is, no player has an incentive to deviate from the protocol at any point.

The key idea in our proof is that at certain points in the execution of the protocol, a player becomes "locked in" to a type (and the random strings r_0, or r_0 and v, in the case of players 0 and 1 respectively). Following this point, the player can no longer fool the other players about its type. Formally, given a vertex u of the game tree, we say that *player i is committed to value x in u* if player i has already executed the following actions in u:

- Agent 0: has sent messages m_1, m_ℓ to players 1 and ℓ (resp.), and the first such messages sent have $m_1 \oplus m_\ell = x$.
- Agent 1: has sent messages m_0, m_2 to players 0 and 2 (resp.), and the first such messages sent have $m_0 \oplus m_2 = x$.
- Agent i for $i \neq 0, 1$: either
  - Agent i has sent at least one message to its parent in the tree, and the first message sent to the parent is x, or
  - Agent i has sent at least one message m to one of its children in the tree, and the contents of the first such message satisfies $m \oplus v = x$.[2]

We point out a subtlety in the definition above: under the protocol, a player $i \notin \{0, 1\}$ is not meant to contact its children in the tree before it has finished executing the parent-child protocol with its own parent. Thus, if i follows the protocol, it commits to its type by revealing its type to its parent, not by sending a message to some child. However, if i decides to deviate, it might try to gain some advantage by contacting its children before it is supposed to, in order to elicit their types from them before it commits to its own value. The children cannot detect this, as they cannot know when i has already revealed its type to its parent in the tree. Nevertheless, i cannot gain by doing so: as soon as i sends a message m to some child, it is effectively "locked in" to the value $m \oplus v$, since v will be revealed at the end of the protocol. Even if the child then responds with its true type, it is too late for i to "change its mind" about the value to which it committed.

We prove that "commitments are real", in the sense that if player i is committed to some value, then this value will eventually appear in the *values* array of some other player, even if player i deviates from the protocol:

▶ **Lemma 6.** *Let u be a leaf that is reached in a run where only player i deviates, and let u' be a vertex on the path to u. Suppose that player i is committed to value x at u'. Then in u, either some player carries out a punishment, or there is a player $j \neq i$ such that $values_j[i] = x$.*

[2] We assume for convenience that the private randomness of the players is fixed at the beginning of the run, so that the value v is defined even if player 1 has not yet taken a step.

Observe that players that do not deviate from the protocol cannot be stopped from committing to their true type, even if some other player deviates: under the protocol, the value to which a player commits does not depend on any messages it receives.

Next, we show that if the protocol completes without any player carrying out a punishment, then even if player i has deviated, all other players agree on their *values* arrays:

▶ **Lemma 7.** *Let u be a leaf that is reached by a run where only player i deviates, and suppose that no player takes a punishment action in u. Then in u, any two players $j \neq k \in [n] \setminus \{i\}$ have $values_j = values_k$.*

Intuitively, this is because the *values* arrays are propagated along the edges of the two-connected subgraph G', so any inconsistency will be detected along a path that does not include player i. Combined with the previous lemma, we now see that if player i is committed to some type, then this type will eventually appear in *all* players' *values* arrays, even if player i deviates (unless some player carries out a punishment, which is never in player i's interest). This means that the value x will be the value used when calling the mediator, and after committing, player i can do nothing to change that, short of deviating from the protocol in an obvious way that would cause some player to punish it. For all players that follow the protocol, their true types will be used (as they commit to those values). It follows that after committing to a type, player i has no incentive to deviate from the protocol, because doing so cannot improve its expected utility.

The next part of our proof deals with deviations that might occur prior to committing to a type. In particular, we must rule out the possibility that a player i that has not yet committed to its type decides to lie, and commit to a value other than its true type. To rule out such deviations, we prove that for each player i, at any point in the run where player i has not committed to its type, its belief about the other players' types is unchanged from the prior (in other words, player i does not gain any information before committing – even if player i deviates from the protocol).

This means that before player i commits to a type, it is effectively in the same situation that it would be in at the beginning of the mediated game: it knows only its own type and the prior distribution of the other players' types. Since the mediator is assumed to be a communication equilibrium, player i has no incentive to lie about its type in this situation. Formalizing this intuition and making its precise is somewhat involved, since player i can deviate in many ways that do not immediately translate to "lying about its type" under the protocol; nevertheless, we show that any strategy that player i might employ can be translated into a distribution on types that player i might send to the mediator in the mediated game, while following the protocol translates to telling the mediator player i's true type. We are therefore able to show that no strategy other than following the protocol can improve player i's expected payoff.

6 Examples of Full-Information Communication Equilibria

We point out two classes of normal-form Bayesian games Γ, such that any communication equilibrium for Γ is also guaranteed to be a *full-information* communication equilibrium (that is, any mediator d that satisfies Definition 2 for Γ also satisfies Definition 3.) The proofs are straightforward, and they are omitted here.

Constrained games. The first class we consider are games where given the behavior of all the other agents, there is only one "good" action that player i can take.

The games in this class involve a set of *legal outcomes* $L \subseteq A_1 \times \ldots \times A_n$, and we require that players prefer to reach a legal outcome above all other considerations. This is referred to as *solution preference* [4], and is assumed by most work on rational distributed computing. Formally, the requirement is that for any player i, type profile $t \in T_1 \times \ldots T_n$, legal outcome $a \in L$ and illegal outcome $a \notin L$, we have $u_i(t, a) \geq u_i(t, a')$.

▶ **Definition 8** (Constrained games). *We say that a set of legal outcomes $L \subseteq A_1 \times \ldots \times A_n$ is* constrained *if for any player i and legal outcome $(a_i, a_{-i}) \in L$, there does not exist any action $a'_i \neq a_i$ such that $(a'_i, a_{-i}) \in L$.*

A game Γ is called constrained *if there exists a set of outcomes $L \subseteq A_1 \times \ldots \times A_n$, such that Γ has the solution-preference property with respect to L, and L is constrained.*

Examples of this class include agreement problems such as leader election and consensus, where the set of legal solutions requires all players to agree on an output, but also other problems, depending on the output specification; we can turn any problem into a constrained game by simply having each node output its neighbors' actions in addition to its own. For instance, computing a spanning tree can be cast as a constrained game, by having the output (action) of each player include both its children and its parent in the tree. This version of the spanning-tree problem makes explicit the intuition that in any legal solution, if node u claims v as its parent, then v should agree that u is its child, and vice-versa.

Games with local utility functions. The second class we consider are games where each player's payoff depends only on its own type and action, that is, for any two type profiles $t, t' \in T$ and action profiles $a, a' \in A$, if $t_i = t'_i$ and $a_i = a'_i$, then $u_i(t, a) = u_i(t', a')$. Examples of games in this class include resource-allocation problems (e.g., dynamic spectrum allocation), where players only care about the resources allocated to them, and do not care about the allocation of the remaining resources to the other players. In local games, revealing the other players' recommended actions does not provide any additional incentive for player i to deviate from the mediator's recommendation; it is therefore not hard to show that any communication equilibrium is also a full-information communication equilibrium.

7 Communication Lower Bound

In this section we show that for some network graphs and games, every protocol that achieves a given full-information equilibrium must send $\Omega(n^2 \cdot b)$ bits in total. This holds even if the actions of the players are "short", requiring b bits to represent.

▶ **Theorem 9.** *For every $n \geq 1, b \geq 1$, there is a $2n$-player normal-form Bayesian game Γ with b-bit types and actions, and a full-information communication equilibrium d for Γ, such that any protocol that implements d must send $\Omega(b \cdot n^2)$ bits in expectation on a ring.*

Proof sketch. Consider the following $2n$-player Bayesian game Γ, where the players are given by $\{0, \ldots, 2n-1\}$, and each player has 2^b possible types, denoted $\left\{0, \ldots, 2^b - 1\right\}$. The types of the players are iid uniformly random. The actions available to each player are also given by $\left\{0, \ldots, 2^b - 1\right\}$.

Given a player $i \in [2n]$, the *opposite player* of i, denoted $-i$, is player $(i+n) \bmod 2n$. We define the utility u_i of player i to be 1 if player i outputs the type t_{-i} of its opposite player, and 0 otherwise. It is not hard to show that for this game, $(1, \ldots, 1)$ is a full-information communication equilibrium, which is achieved by having the mediator tell each player the opposite player's type.

Now consider a distributed implementation of this equilibrium. Suppose the $2n$ players are arranged consecutively in a ring, and consider only *synchronous* executions, where the scheduler lets the players run in round-robin order, and every message that is sent by player i to player j is delivered the next time player j takes a step. Fix a protocol P which achieves utility $(1,\ldots,1)$ in all runs (in fact, our proof can be extended to protocols that achieve utility $(1,\ldots,1)$ with constant probability, see Appendix A). Every run of P must end with player i correctly outputting the type of the opposite player, t_{-i}. Intuitively, this means b bits of information must flow from $-i$ to i, and they must be repeated along every edge between $-i$ and i; the distance between players i and $-i$ is n, so the total number of bits sent is $\Omega(n \cdot b)$, and these bits only "help" players $i, -i$, so every other pair of players must also send $\Omega(n \cdot b)$ bits of their own. The total communication complexity is therefore $\Omega(n^2 \cdot b)$.

The formal argument is given in Appendix A. It uses the technique of [11], where we consider each balanced cut in the graph, and argue by reduction to two-party communication complexity that $\Omega(n \cdot b)$ bits must flow across the cut. Since there are $\Theta(n)$ balanced cuts in the ring, and each edge appears in exactly one, the total communication is $\Omega(n^2 \cdot b)$ bits. ◀

8 Necessity of Two-Connectivity

Finally, we show that there is some full-information communication equilibrium that cannot be implemented in any network that is not 2-connected. We start from a two-player game, as follows:

▶ **Theorem 10.** *There is a normal-form Bayesian two-player game Γ, which has a welfare-maximizing full-information communication equilibrium d, such that no asynchronous protocol P implements d or any other welfare-maximizing communication equilibrium.*

(Recall that a *welfare-maximizing* equilibrium is one where the expected sum of the players' utilities is maximized.)

Theorem 10 implies that for any graph G that is not 2-connected, there is a normal-form Bayesian game Γ' such that no protocol can achieve maximum-welfare in Γ': since G is not 2-connected, it has a *bridge*, an edge (u, v) whose removal disconnects the graph. We take Γ' to be the game where players u, v take on the roles of the two players in Γ, and all other players have only one possible type and action. It is not hard to see that the impossibility result of Theorem 10 carries over to Γ'.

The proof of Theorem 10 is inspired by a game from [7, Ch. 4], and essentially involves a game of "chicken": the utility function incentivizes each player to learn the other player's type, but not to reveal its own true type. However, the only welfare-maximizing equilibrium is achieved when both players reveal their true types to one another. In any run of an asynchronous protocol, some player must be the first to reveal some information about its type, but neither player wants to "go first", because revealing this information gives an advantage to the other player, and decreases the revealing player's expected payoff. Therefore, there is no protocol that implements the welfare-maximizing equilibrium: players always have an incentive to deviate from the protocol. There is some subtlety involved in capturing what it means to "reveal information about the type" (for example, suppose player 1 sends its true type with probability $1/100$, and otherwise sends a random type), and proving that this indeed gives the other player an advantage. See Appendix B for the details.

References

1 Ittai Abraham, Danny Dolev, Ivan Geffner, and Joseph Y. Halpern. Implementing mediators with asynchronous cheap talk. In *PODC 2019*, pages 501–510, 2019.

2 Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: Robust mechanisms for rational secret sharing and multiparty computation. In *PODC 2006*, pages 53–62, 2006.

3 Ittai Abraham, Danny Dolev, and Joseph Y Halpern. Lower bounds on implementing robust and resilient mediators. In *TCC 2008*, pages 302–319, 2008.

4 Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. Distributed protocols for leader election: A game-theoretic perspective. *ACM Trans. Economics and Comput.*, 7(1):4:1–4:26, 2019.

5 Yehuda Afek, Yehonatan Ginzberg, Shir Landau Feibish, and Moshe Sulamy. Distributed computing building blocks for rational agents. In *PODC 2014*, pages 406–415, 2014.

6 Yehuda Afek, Shaked Rafaeli, and Moshe Sulamy. The role of a-priori information in networks of rational agents. In *DISC 2018*, pages 5:1–5:18, 2018.

7 Mor Amitai. Cheap-talk with incomplete information on both sides. Discussion paper 90, Center for Rationality, The Hebrew University of Jerusalem, 1996.

8 Itai Ashlagi, Dov Monderer, and Moshe Tennenholtz. On the value of correlation. *J. Artif. Int. Res.*, 33(1):575–613, 2008.

9 Robert J. Aumann. Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, 1(1):67–96, 1974.

10 Elchanan Ben-Porath. Cheap talk in games with incomplete information. *J. Economic Theory*, 108(1):45–71, 2003.

11 Arkadev Chattopadhyay, Jaikumar Radhakrishnan, and Atri Rudra. Topology matters in communication. In *FOCS 2014*, pages 631–640, 2014.

12 Francoise Forges. An approach to communication equilibria. *Econometrica*, 54(6):1375–1385, 1986.

13 Joseph Halpern and Vanessa Teague. Rational secret sharing and multiparty computation. In *STOC 2004*, pages 623–632, 2004.

14 Joseph Y Halpern and Xavier Vilaça. Rational consensus. In *PODC 2016*, pages 137–146, 2016.

15 Itay Harel, Amit Jacob-Fanani, Moshe Sulamy, and Yehuda Afek. Consensus in Equilibrium: Can One Against All Decide Fairly? In *OPODIS 2019*, pages 20:1–20:17, 2019.

16 Juraj Hromkovič, Claus-Dieter Jeschke, and Burkhard Monien. Optimal algorithms for dissemination of information in some interconnection networks. *Algorithmica*, 10(1):24–40, 1993.

17 Sergei Izmalkov, Silvio Micali, and Matt Lepinski. Rational secure computation and ideal mechanism design. In *FOCS 2015*, pages 585–594, 2015.

18 David M Kreps and Robert Wilson. Sequential equilibria. *Econometrica: Journal of the Econometric Society*, pages 863–894, 1982.

19 Matt Lepinski, Silvio Micali, Chris Peikert, and Abhi Shelat. Completely fair sfe and coalition-safe cheap talk. In *PODC 2004*, pages 1–10, 2004.

20 Merav Parter and Eylon Yogev. Secure distributed computing made (nearly) optimal. In *PODC 2019*, pages 107–116, 2019.

21 Ryan M. Rogers and Aaron Roth. Asymptotically truthful equilibrium selection in large congestion games. In *Proceedings of the Fifteenth ACM Conference on Economics and Computation*, EC '14, pages 771–782, 2014.

22 R. W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2:65–67, 1973.

23 Assaf Yifrach and Yishay Mansour. Fair leader election for rational agents in asynchronous rings and networks. In *PODC 2018*, pages 217–226, 2018.

A Missing Details from the Communication Lower Bound

We prove that any protocol implementing the equilibrium $(1,\dots,1)$ in the game from Section 7 must send $\Omega(n^2 \cdot b)$ bits in total (in expectation). In fact, this holds even if the protocol only achieves global output $(1,\dots,1)$ with probability $1-\epsilon$, where $\epsilon \in (0,1)$ is some constant error probability (which may exceed $1/2$).

Proof. Consider the family of cuts $\mathcal{S} = \{S_j\}_{j\in[n]}$, where S_j is the cut containing edges $\{j, (j+1) \bmod 2n\}$ and the "opposite edge" $\{(j+n) \bmod 2n, (j+n+1) \bmod 2n\}$. We argue that across each such cut, $\Omega(nb)$ bits of communication must flow.

Let A_j, B_j be the players on the two sides of the cut S_j, and let $\mathbf{M}_j$ be a random variable representing the messages that flow across the two edges in the cut S_j. Given a tuple X of players, let t_X represent the tuple consisting of the types of all players $i \in X$.

Consider a game $\mathcal{G}_j$ between two parties, Alice and Bob, where Alice receives the types $\mathbf{t}_{A_j}$ of all players on one side of the cut, and Bob receives the types $\mathbf{t}_{B_j}$ of all players on the other side of the cut. The goal is for Alice and Bob to each output the other party's input.

For information-theoretic reasons, in order for Alice to output $\mathbf{t}_{A_j}$ correctly with probability $1-\epsilon$, Bob must send her $\Omega(n \cdot b)$ bits in expectation: the input of each player i is a uniformly random string of length $n \cdot b$, so the entropy of $\mathbf{t}_{B_j}$ is $\mathbb{H}(\mathbf{t}_{B_j}) = nb$. However, by Fano's inequality, conditioned on the communication $\mathbf{M}_j$ between Alice and Bob, we have

$$\mathbb{H}(\mathbf{t}_{B_j}|\mathbf{M}_j) \le \mathbb{H}(\epsilon) + \epsilon \cdot \log(2^{nb}-1) < 1 + \epsilon nb.$$

(Here, $\mathbb{H}(\epsilon) = -\epsilon\log(\epsilon) - (1-\epsilon)\log(1-\epsilon)$ is the binary entropy of $\epsilon \in (0,1)$, which satisfies $\mathbb{H}(\epsilon) \in (0,1)$.) For symmetric reasons, we also have $\mathbb{H}(\mathbf{t}_{A_j}|\mathbf{M}_j) \le 1 + \epsilon nb$. This means that $\mathbb{I}(\mathbf{M}_j;\mathbf{t}) = \mathbb{H}(\mathbf{t}_{A_j}\mathbf{t}_{B_j}) - \mathbb{H}(\mathbf{t}_{A_j}\mathbf{t}_{B_j}|\mathbf{M}_j) = 2nb - 2\epsilon nb - 2 = \Omega(nb)$, assuming n is sufficiently large. Mutual information is symmetric: we can also write $\mathbb{I}(\mathbf{M}_j;\mathbf{t}) = \mathbb{H}(\mathbf{M}_j) - \mathbb{H}(\mathbf{M}_j|\mathbf{t}) \le \mathbb{H}(\mathbf{M}_j)$, which implies that $\mathbb{H}(\mathbf{M}_j) \ge 2nb$. Entropy is never greater than the expected number of bits required to represent $\mathbf{M}_j$, and the claim follows.

Alice and Bob can *simulate* the protocol P to win the game $\mathcal{G}_j$: each party locally simulates the vertices on their side of the cut, and sends to the other party the messages crossing the cut edges. Under P, with probability at least $1-\epsilon$, each vertex outputs the type of the opposite vertex, as this is the only scenario where the utilities of the players are $(1,\dots,1)$. Thus, with probability $1-\epsilon$, Alice and Bob learn the correct output for $\mathcal{G}_j$ by simulating P. The communication between the two parties in this simulation is exactly $\mathbf{M}_j$.

Observe that each edge of the ring appears in exactly one cut in the family $\mathcal{S} = \{S_j\}_{j\in[n]}$. Thus, by linearity of expectation (summing over all cuts), the total communication that P sends on all edges is, in expectation, $\Omega(n^2 \cdot b)$. ◀

B Necessity of Two-Connectivity: Proof Overview

In this section we give a detailed overview of the proof of Theorem 10.

Consider the following 2-player Bayesian game Γ: there are two possible types, $\{1,2\}$, and the type of each player is chosen uniformly and independently of the other player. The game has two possible actions, denoted $\{1,2\}$ (the same as the set of types). The utilities are given by the following matrices, $\left\{u^{i,j} \;\middle|\; i,j \in \{1,2\}\right\}$, where element (a_1,a_2) of matrix $u^{i,j}$ represents the utilities of the two players when their types are i,j (respectively) and the actions they take are a_1,a_2 (respectively).

$$u^{1,1} = \begin{pmatrix} 1,1 & 2,-2 \\ -2,2 & 0,0 \end{pmatrix},\ u^{1,2} = \begin{pmatrix} 0,0 & 0,0 \\ 1,1 & 0,0 \end{pmatrix},\ u^{2,1} = \begin{pmatrix} 0,0 & 1,1 \\ 0,0 & 0,0 \end{pmatrix},\ u^{2,2} = \begin{pmatrix} 0,0 & -2,2 \\ 2,-2 & 1,1 \end{pmatrix}.$$

It is not difficult to verify that the information-dissemination mediator d_{info} defined in Section 3 is a full-information communication equilibrium for Γ (i.e., it satisfies the conditions of Definition 3), and furthermore, it is the only communication equilibrium that achieves social welfare of 2 (the *social welfare* of an equilibrium with utilities $(u_1, \ldots, u_n)$ in an n-player game is the sum $\sum_{i=1}^{n} u_i$ of the players' utilities).

Now fix a protocol P which achieves utility $(1,1)$ in all runs, and let us show that P *cannot* be an equilibrium.

While we have so far avoided giving the formal definitions associated with extensive-form games with imperfect information, here we cannot avoid them completely. However, to simplify matters, we consider a restricted set of executions, and give only the simplified definitions that are necessary to understand the proof. Throughout, we use bold-face letters to denote random variables, and plain letters to denote concrete values.

We consider only runs of P resulting from a scheduler that schedules the two players in alternating order, and immediately delivers every message that is sent (e.g., if player 1 sends a message to player 2, then in the next step the scheduler immediately schedules player 2 and delivers the message). We also assume w.l.o.g. that both players send a message every time they take a step, as any protocol that does not do this can be transformed into one that does, without affecting whether or not the protocol is an equilibrium (under this specific set of runs). The history h of such a run is represented by the sequence of messages sent and received by the two players; both players know the entire history, since they know what they sent and what they received. Moreover, since the types are initially independent, they remain independent conditioned on any history[3]. We emphasize that a *run* consists of the types of the two players and all the steps they have taken (messages sent and received), while a *history* consists only of the steps taken, as those are visible to both players.

After fixing the scheduler, for every assignment of types, the protocol P induces a probability distribution over histories of every given length. The *belief* of player 1 about player 2's type given the history h (and similarly for player 2) is the distribution $p_1(\cdot|t_1, h) : \{1,2\} \to [0,1]$, where

$$p_1(\mathbf{t}_2 = j | t_1, h) = \frac{\Pr\left[\mathbf{t}_2 = j | \mathbf{t}_1 = t_1, \mathbf{h} = h\right]}{\Pr\left[\mathbf{t}_2 = j | \mathbf{t}_1 = t_1\right]}.$$

The probability is taken with respect to the protocol's distribution over histories of length $|h|$ and the random assignment of types. However, since the player's types are independent, and they remain so conditioned on the history, we can omit t_1 from our notation and write $p_1(\cdot|h)$ (and for player 2, we omit t_2 and write $p_2(\cdot|h)$). By Bayes' law, we can also write

$$p_1(\mathbf{t}_2 = j | h) = \frac{\Pr\left[\mathbf{h} = h | \mathbf{t}_2 = j\right]}{2 \Pr\left[\mathbf{h} = h\right]},$$

where the probability is again with respect to the protocol's distribution over histories of length $|h|$. In particular, $p_1(\mathbf{t}_2 = j | h) > 0$ iff there exists a run r where $\mathbf{t}_2 = j$ and the history is h.

[3] It is well-known that no communication protocol can create dependence between its inputs if they were initially independent.

Now we are ready to show that P cannot be an equilibrium. We begin by observing that since P is welfare-maximizing, every run must end with each player outputting the other player's type, to reach the maximum total payoff of 2. Thus, if P has a run with types (t_1, t_2) that ends after some history h, then

$$p_2(\mathbf{t}_1 = t_1|h) = 1, \qquad p_1(\mathbf{t}_2 = t_2|h) = 1.$$

Next, we show that at any history h where $p_2(\mathbf{t}_1 = a|h) > 0$ (that is, "player 2 still believes that $t_1 = a$ is possible"), player 1 can force player 2 to output a by deviating from the protocol (and vice-versa):

▶ **Observation 11.** *Let r be a run with a history h, and let $a \in \{1, 2\}$ be a type such that $p_2(\mathbf{t}_1 = a|h) > 0$. Then there is a strategy s_1 for player 1 starting from r, such that in every extension from r where player 2 plays according to P and player 1 plays s_1, player 2 always outputs a. The same holds with the roles of players 1 and 2 reversed.*

Proof. Since $p_2(\mathbf{t}_1 = a|h) > 0$, there exists a run r where $\mathbf{t}_1 = a$ and the history is h. In every extension of r where both players follow the protocol, player 2 eventually outputs a, as we assumed that the protocol always terminates with each player outputting the other player's type. Therefore, player 1's strategy s_1 is to simply behave the same way that it would under P when $\mathbf{t}_1 = a$, regardless of its true type, as this will always end with player 2 outputting a (if player 2 follows P). ◀

Now consider a run r of P, where the types of both players are 1. Observe that when a player *sends* a message, this does not change its belief; only receiving a message can change a player's belief about the other player's type.

The run begins with the prior $p_1(\mathbf{t}_2 = 1) = p_2(\mathbf{t}_1 = 1) = 1/2$, and it ends at a history h where $p_1(\mathbf{t}_2 = 1|h) = p_2(\mathbf{t}_1 = 1|h) = 1$. Let h be the *longest* history during r such that $p_1(\mathbf{t}_2 = 1|h) = p_2(\mathbf{t}_1 = 1|h) = 1/2$. Let h' be the history following h in r; then either $p_1(\mathbf{t}_2 = 1|h') \neq 1/2$ or $p_2(\mathbf{t}_1 = 1|h') \neq 1/2$, and we assume w.l.o.g. that $p_1(\mathbf{t}_2 = 1|h') > 1/2$. This implies that $p_2(\mathbf{t}_1 = 1|h') = 1/2$, because at each step, only one player's belief changes (the player that *receives* a message).

Since $p_2(\mathbf{t}_1 = 2|h') = 1 - p_2(\mathbf{t}_1 = 1|h') = 1/2$, by Observation 11, there is a strategy s_1 for player 1 from h' that always leads to player 2 outputting 2. Player 1's expected payoff is improved by following this strategy and outputting 1: under P, player 1 always receives a payoff of 1, but if player 1 follows s_1 and outputs 1, the expected payoff is

$$\begin{aligned} &p_1(\mathbf{t}_2 = 1|h') \cdot u^{1,1}(1, 2) + p_1(\mathbf{t}_2 = 2|h') \cdot u^{1,2}(1, 2) \\ &\qquad = p_1(\mathbf{t}_2 = 1|h') \cdot 2 + p_1(\mathbf{t}_2 = 2|h') \cdot 0 > \frac{1}{2} \cdot 2 = 1. \end{aligned}$$

Note that the strategy s_1 is not the strategy player 1 is supposed to follow under P, since playing according to P always ends with both players earning 1. Therefore, at h' player 1's rational choice is not to follow P, meaning P is not an equilibrium.

In Search for an Optimal Authenticated Byzantine Agreement

Alexander Spiegelman ✉
Novi Research, Menlo Park, USA

Abstract

In this paper, we challenge the conventional approach of state machine replication systems to design deterministic agreement protocols in the eventually synchronous communication model. We first prove that no such protocol can guarantee bounded communication cost before the global stabilization time and propose a different approach that hopes for the best (synchrony) but prepares for the worst (asynchrony). Accordingly, we design an *optimistic* byzantine agreement protocol that first tries an efficient deterministic algorithm that relies on synchrony for termination only, and then, only if an agreement was not reached due to asynchrony, the protocol uses a randomized asynchronous protocol for fallback that guarantees termination with probability 1.

We formally prove that our protocol achieves optimal communication complexity under all network conditions and failure scenarios. We first prove a lower bound of $\Omega(ft + t)$ for synchronous deterministic byzantine agreement protocols, where t is the failure threshold, and f is the actual number of failures. Then, we present a tight upper bound and use it for the synchronous part of the optimistic protocol. Finally, for the asynchronous fallback, we use a variant of the (optimal) VABA protocol, which we reconstruct to safely combine it with the synchronous part.

We believe that our adaptive to failures synchronous byzantine agreement protocol has an independent interest since it is the first protocol we are aware of which communication complexity optimally depends on the actual number of failures.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security

Keywords and phrases Byzantine agreement, Optimistic, Asynchronous fallback

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.38

Related Version *Full Version*: https://arxiv.org/pdf/2002.06993.pdf

1 Introduction

With the emergence of the Blockchain use case, designing efficient geo-replicated Byzantine tolerant state machine replication (SMR) systems is now one of the most challenging problems in distributed computing. The core of every Byzantine SMR system is the Byzantine agreement problem (see [3] for a survey), which was first introduced four decades ago [33] and has been intensively studied since then [11, 22, 26, 24]. The bottleneck in geo-replicated SMR systems is the network communication, and thus a substantial effort in recent years was invested in the search for an optimal communication Byzantine agreement protocol [20, 38, 10, 30].

To circumvent the FLP [17] result that states that deterministic asynchronous agreement protocols are impossible, most SMR solutions [12, 20, 38, 23] assume eventually synchronous communication models and provide safety during asynchronous periods but can guarantee progress only after the global stabilization time (GST).

Therefore, it is quite natural that state-of-the-art authenticated Byzantine agreement protocols [20, 38, 10, 30] focus on reducing communication cost after GST, while putting up with the potentially unbounded cost beforehand. For example, Zyzzyva [23] and later SBFT [20] use threshold signatures [34] and collectors to reduce the quadratic cost induced by the all-to-all communication in each view of the PBFT [12] protocol. HotStuff [38] leverages

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 38; pp. 38:1–38:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ideas presented in Tendermint [10] to propose a linear view-change mechanism, and a few follow-up works [30, 31, 9] proposed algorithms for synchronizing parties between views. Some [30, 31] proposed a synchronizer with a linear cost after GST in failure-free runs, while others [9] provided an implementation that guarantees bounded memory even before GST. However, none of the above algorithms bounds the number of views executed before GST, and thus none of them can guarantee a bounded total communication cost.

We argue in this paper that designing agreement algorithms in the eventually synchronous model is not the best approach to reduce the total communication complexity of SMR systems and propose an alternative approach. That is, we propose to forgo the eventually synchronous assumptions and instead optimistically consider the network to be synchronous and immediately switch to randomized asynchronous treatment if synchrony assumption does not hold. Our goal in this paper is to develop an *optimistic* protocol that adapts to network conditions and actual failures to guarantee termination with an optimal communication cost under all failure and network scenarios.

1.1 Contribution

Vulnerability of the eventually synchronous model. A real network consists of synchronous and asynchronous periods. From a practical point of view, if the synchronous periods are too short, no deterministic Agreement algorithm can make progress [17]. Therefore, to capture the assumption that eventually there will be a long enough synchronous period for a deterministic Agreement to terminate, the eventually synchronous model assumes that every execution has a point, called GST, after which the network is synchronous. In our first result, we capture the inherent vulnerability of algorithms designed in the eventually synchronous communication model. That is, we exploit the fact that GST can occur after an arbitrarily long time to prove the following lower bound:

▶ **Theorem 1.** *There is no eventually synchronous deterministic Byzantine agreement protocol that can tolerate a single failure and guarantee bounded communication cost even in failure-free runs.*

Tight bounds for synchronous Byzantine agreement. To develop an optimal optimistic protocol that achieves optimal communication under all failure and network scenarios we first establish what is the best we can achieve in synchronous settings. Dolev and Reischuk [14] proved that there is no deterministic protocol that solves synchronous Byzantine agreement with $o(t^2)$ communication cost, where t is the failure threshold. We generalize their result by considering the actual number of failures $f \leq t$ and prove the following lower bound:

▶ **Theorem 2.** *Any synchronous deterministic Byzantine agreement protocol has $\Omega(ft + t)$ communication complexity.*

It is important to note that the lower bound holds even for deterministic protocols that are allowed to use perfect cryptographic schemes such as threshold signatures and authenticated links. Then, we present the first deterministic cryptography-based synchronous Byzantine agreement protocol that matches our lower bound for the authenticated case. That is, we prove the following:

▶ **Theorem 3.** *There is a deterministic synchronous authenticated Byzantine agreement protocol with $O(ft + t)$ communication complexity.*

We believe these results are interesting on their own since they are the first to consider the actual number of failures, which was previously considered in the problem of early decision/stopping [15, 21], for communication complexity analysis of the Byzantine agreement problem.

Optimal optimistic Byzantine agreement. Our final contribution is an optimistic Byzantine agreement protocol that tolerates up to $t < n/3$ failures and has asymptotically optimal communication cost under all network conditions and failure scenarios. That is, we prove the following:

▶ **Theorem 4.** *There is an authenticated Byzantine agreement protocol with $O(ft + t)$ communication complexity in synchronous runs and expected $O(t^2)$ communication complexity in all other runs.*

To achieve the result, we combine our optimal adaptive synchronous protocol with an asynchronous fallback, for which we use a variant of VABA [1]. As we shortly explain, the combination is not trivial since we need to preserve safety even if parties decide in different parts of the protocol, and implement an efficient mechanism to prevent honest parties from moving to the fallback in synchronous runs.

1.2 Technical overview

The combination of our synchronous part with the asynchronous fallback introduces two main challenges. The first challenge is to design a mechanism that (1) makes sure parties do not move to the fallback unless necessary for termination, and (2) has $O(ft + t)$ communication complexity in synchronous runs. The difficulty here is twofold: first, parties cannot always distinguish between synchronous and asynchronous runs. Second, they cannot distinguish between honest parties that complain that they did not decide (due to asynchrony) in the first part and Byzantine parties that complain because they wish to increase the communication cost by moving to the asynchronous fallback. To deal with this challenge, we implement a *Help&tryHalting* procedure. In a nutshell, parties try to avoid the fallback part by helping complaining parties learn the decision value and move to the fallback only when the number of complaints indicates that the run is not synchronous. This way, each Byzantine party in a synchronous run cannot increase the communication cost by more than $O(n) = O(t)$, where n is the total number of parties.

The second challenge in the optimistic protocol is to combine both parts in a way that guarantees safety. That is, since some parties may decide in the synchronous part and others in the asynchronous fallback, we need to make sure they decide on the same value. To this end, we use the *leader-based view (LBV)* abstraction, defined in [37], as a building block for both parts. The LBV abstraction captures a single view in a view-by-view agreement protocol such that one of its important properties is that a sequential composition of them preserves safety. For optimal communication cost, we adopt techniques from [38] and [1] to implement the LBV abstraction with an asymptotically linear cost ($O(n)$).

Our synchronous protocol operates up to n sequentially composed pre-defined linear LBV instances, each with a different leader. To achieve an optimal (adaptive to the number of actual failures) cost, leaders invoke their LBVs only if they have not yet decided. In contrast to eventually synchronous protocols, the synchronous part is designed to provide termination only in synchronous runs. Therefore, parties do not need to be synchronized before views, but rather move from one LBV to the next at pre-defined times. As for the asynchronous fallback, we use the linear LBV building block to reconstruct the VABA [1] protocol in a way that forms a sequential composition of LBVs, which in turn allows a convenient sequential composition with the synchronous part.

1.3 Related work

The idea of combining several agreement protocols is not new. The notion of speculative linearizability [19] allows parties to independently switch from one protocol to another, without requiring them to reach agreement to determine the change of a protocol. Aguilera and Toueg [2] presented an hybrid approach to solve asynchronous crash-fault consensus by combining randomization and unreliable failure detection. Guerraoui et al [18] defined an abstraction that captures byzantine agreement protocols and presented a framework to compose several such instances.

Some previous work on Byzantine agreement consider a fallback in the context of the number rounds required for termination [7, 27, 35]. That is, in well-behaved runs parties decide in a single communication round, wheres in all other runs they fallback to a mode that requires more rounds to reach an agreement. We, in contrast, are interested in communication complexity. To the best of our knowledge, our protocol is the first protocol that adapts its communication complexity based on the actual number of failures.

The combination of synchronous and asynchronous runs in the context of Byzantine agreement was previously studied by Blum et al. [5]. Their result is complementary to ours since they deal with optimal resilience rather than optimal communication. They showed lower and upper bounds on the number of failures that both (synchronous and asynchronous) parts can tolerate. For the lower bound, they showed that $t_a + 2t_s < n$, where t_a and t_s is the threshold failure in asynchronous and synchronous runs, respectively. In our protocol $t_a = t_s < n/3$, which means that the protocol is optimal in the sense that neither t_a or t_s can be increased without decreasing the other. For the upper bound, they present a matching algorithm for any t_a and t_s that satisfy the weak validity condition. Our protocol, in contrast, satisfy the more practical external validity condition (see more details in the next section) with an optimal communication cost.

As for asynchronous Byzantine agreement, the lower bound in [1] shows that there is no protocol with optimal resilience and $o(n^2)$ communication complexity. Two recent works by Cohen et al. [13] and Blum et al [4]. circumvent this lower bound by trading optimal resilience. That is, their protocols tolerate $f < (1 - \epsilon)n/3$ Byzantine faults. We consider in this paper optimal resilience and thus our protocol achieves optimal communication complexity in asynchronous runs.

The use of cryptographic tools (e.g. PKI and threshold signatures schemes) is very common in distributed computing to reduce round and communication complexity. To be able to focus on the distributed aspect of the problem, many previous algorithms assume ideal cryptographic tools to avoid the analysis of the small error probability induced by the security parameter. This includes the pioneer protocols for Byzantine broadcast [16, 14] and binary asynchronous Byzantine agreement [6], recent works on synchronous Byzantine agreement [29, 32], and most of the exciting practical algorithms [23, 12] including the state-of-the-art communication efficient ones [12, 38, 20, 10]). We follow this approach and assume ideal threshold signatures schemes for better readability.

2 Model

Following practical solutions [12, 20, 38, 23, 28], we consider a Byzantine message passing peer to peer model with a set Π of n parties and a computationally bounded adversary that corrupts up to $t < n/3$ of them, $O(t) = O(n)$. Parties corrupted by the adversary are called *Byzantine* and may arbitrarily deviate from the protocol. Other parties are *honest*. To strengthen the result we consider an adaptive adversary for the upper bound and static adversary for the lower bound. The difference is that a *static* adversary must decide what parties to corrupt at the beginning of every execution, whereas an *adaptive* adversary can choose during the executions.

Communication and runs. The communication links are reliable but controlled by the adversary, i.e., all messages sent among honest parties are eventually delivered, but the adversary controls the delivery time. We assume a known to all parameter Δ and say that a run of a protocol is *eventually synchronous* if there is a *global stabilization time (GST)* after which all message sent among honest parties are delivered within Δ time. A run is *synchronous* if GST occurs at time 0, and *asynchronous* if GST never occurs.

The Agreement problem. Each party get an input value from the adversary from some domain $\mathbb{V}$ and the Agreement problem exposes an API to *propose* a value and to output a *decision*. We are interested in protocols that never compromise safety and thus require the following property to be satisfied in all runs:

- Agreement: All honest parties that decide, decide on the same value.

Due to the FLP result [17], no deterministic agreement protocol can provide safety and liveness properties in all asynchronous runs. Therefore, in this paper, we consider protocols that guarantee (deterministic) termination in all synchronous and eventually synchronous runs, and provides a probabilistic termination in asynchronous ones:

- Termination: All honest parties eventually decide.
- Probabilistic-Termination: All honest parties decide with probability 1.

As for validity, honest parties must decide only on values from some domain $\mathbb{V}$. For the lower bounds, to strengthen them as much as possible, we consider the binary case, which is the weakest possible definition:

- Binary validity: The domain of valid values $\mathbb{V} = \{0, 1\}$, and if all honest parties propose the same value $v \in \mathbb{V}$, than no honest party decides on a value other than v.

For the upper bounds, we are interested in practical multi-valued protocols. In contrast to binary validity, in a multi-valued Byzantine agreement we need also to define what is a valid decision in the case that not all parties a priori agree (i.e., propose different values). One option is Weak Validity [33, 5], which allows parties to agree on a pre-defined $\perp$ in that case. This definition is well defined and makes sense for some use cases. When Pease et al. [33] originally defined it, they had in mind a spaceship cockpit with 4 sensors that try to agree even if one is broken (measures a wrong value). However, as Cachin et al, explain in their paper [11] and book [25], this definition is useless for SMR (and Blockchains) since if parties do not a priori agree, then they can keep agreeing on $\perp$ forever leaving the SMR with no "real" progress.

To solve the limitation of being able to agree on $\perp$, we consider the external validity property that was first defined by Cachin et al. [11], which is implicitly or explicitly considered in most practical Byzantine agreement solutions we are aware of [1, 12, 38, 20, 23]. Intuitively, with external validity, parties are allowed to decide on a value proposed by any party (honest and Byzantine) as long as it is valid by some external predicate (e.g., all transaction are valid in the block). To capture the above, we give a formal definition below.

- External validity: The domain of valid values $\mathbb{V}$ is unknown to honest parties. At the beginning of every run, each honest party gets a value v with a proof σ that $v \in \mathbb{V}$ such that all other honest parties can verify.

Note that our definition rules out trivial solutions such as simply deciding on some pre-defined externally valid value because the parties do not know what is externally valid unless they see a proof.

We define an *optimistic Agreement protocol* to be a protocol that guarantees Agreement and External validity in all runs, Termination in all synchronous and eventually synchronous runs, and Probabilistic-Termination in asynchronous runs.

Cryptographic assumptions. We assume a computationally bounded adversary and a trusted dealer that equips parties with cryptographic schemes. Following a common standard in distributed computing and for simplicity of presentation (avoid the analysis of security parameters and negligible error probabilities), we assume that the following cryptographic tools are perfect:

- **Authenticated link.** If an honest party p_i delivers a messages m from an honest party p_j, then p_j previously sent m to p_i.
- **Threshold signatures scheme.** We assume that each party p_i has a private function *share-sign*$_i$, and we assume 3 public functions: *share-validate*, *threshold-sign*, and *threshold-validate*. Informally, given "enough" valid shares, the function *threshold-sign* returns a valid threshold signature. For our algorithm, we sometimes require "enough" to be $t+1$ and sometimes $n-t$. A formal definition is given in the fullpaper [36].

We note that perfect cryptographic schemes do not exist in practice. However, since in real-world systems they often treated as such, we believe that they capture just enough in order to be able to focus on the distributed aspect of the problem. Moreover, all the lower bounds in this paper hold even if protocols can use perfect cryptographic schemes. Thus, the upper bounds are tight in this aspect.

Communication complexity. We denote by f the actual number of corrupted parties in a given run and we are interested in optimistic protocols that utilize f and the network condition to reduce communication cost. Similarly to [1], we say that a *word* contains a constant number of signatures and values, and each message contains at least 1 word. The *communication cost of a run r* is the number of words sent in messages by honest parties in r. For every $0 \leq f \leq t$, let R_f^s and R_f^{es} be the sets of all synchronous and eventually synchronous runs with f corrupted parties, respectively. The *synchronous and eventually synchronous communication cost with f failures* is the maximal communication cost of runs in R_f^s and R_f^{es}, respectively. We say that the *synchronous communication cost of a protocol A* is $G(f,t)$ if for every $0 \leq f \leq t$, its synchronous communication cost with f failures is $G(f,t)$. The *asynchronous communication cost of a protocol A* is the expected communication cost of an asynchronous run of A.

3 Lower Bounds

We present two lower bounds on the communication complexity of deterministic Byzantine agreement protocols in synchronous and eventually synchronous runs. For space limitation, the proof of the following lemma appears to Appendix A.

▶ **Theorem 1** (restated). *There is no eventually synchronous deterministic Byzantine agreement protocol that can tolerate a single failure and guarantee bounded communication cost even in failure-free runs.*

We next prove a lower bound that applies even to synchronous Byzantine agreement algorithms and is adaptive to the number of actual failures f. The proof is a generalization of the proof in [14], which has been proved for the Byzantine broadcast problem and considered the worst-case scenario ($f = t$). It is important to note that the proof captures deterministic authenticated algorithms even if they are equipped with perfect cryptographic tools.

The proof of the following Claim is straight forward and for space limitation is omitted.

▷ Claim 5. The synchronous communication cost with 0 failures of any Byzantine agreement algorithm is at least t.

The following Lemma shows that if honest parties send $o(ft)$ messages, then Byzantine parties can prevent honest parties from getting any of them.

▶ **Lemma 6.** *Assume that there is a Byzantine agreement algorithm A, which synchronous communication cost with f failures is $o(ft)$ for some $1 \leq f \leq \lfloor t/2 \rfloor$. Then, for every set $S \subset \Pi$ of f parties and every set of values proposed by honest parties, there is a synchronous run r' s.t. some honest party $p \in S$ does not get any messages in r'.*

Proof. Let $r \in R^s_f$ be a run in which all parties in S are Byzantine that (1) do not send messages among themselves, and (2) ignore all messages they receive and act like honest parties that get no messages. By the assumption, there is a party $p \in S$ that receives less than $t/2$ messages from honest parties in r. Denote the set of (honest) parties outside S that send messages to p in r by $P \subset \Pi \setminus S$ and consider the following run r':

- Parties in $S \setminus \{p\}$ are Byzantine that act like in r.
- Parties in P are Byzantine. They do not send messages to p, but other than that act as honest parties.
- All other parties, including p, are honest.

First, note that the number of Byzantine parties in r' is $|S| - 1 + |P| \leq f - 1 + t/2 \leq t$. Also, since p acts in r as an honest party that does not receive messages, and all Byzantine parties in r' act towards honest parties in r' ($\Pi \setminus (S \cup P)$) in exactly the same way as they do in r, then honest parties in r' cannot distinguish between r and r'. Thus, since they do not send messages to p in r they do not send in r' as well. Therefore, p does not get any message in r'. ◀

The next Lemma is proven by showing that honest parties that do not get messages cannot safely decide. Not that the case of $f > t/2$ is not required to conclude Theorem 2 since in this case $o(ft) = o(t^2)$.

▶ **Lemma 7.** *For any $1 \leq f \leq \lfloor t/2 \rfloor$, there is no optimistic Byzantine agreement algorithm which synchronous communication cost with f failures is $o(ft)$.*

Proof. Assume by a way of contradiction such protocol A which synchronous communication cost with f failures is $o(ft)$ for some $1 \leq f \leq \lfloor t/2 \rfloor$. Pick a set of $S_1 \subset \Pi$ of f parties and let V be the set of values that honest parties propose. By Lemma 6, there is a run r_1 of A in which honest parties propose values from V s.t. some honest party $p_1 \in S$ does not get any messages. Now let $S_2 = \{p\} \cup S_1 \setminus \{p_1\}$ s.t. $p \in \Pi \setminus S_1$. By Lemma 6 again, there is a run r_2 of A in which honest parties propose values from V s.t. some honest party $p_2 \neq p_1$ does not get any messages. Since $f \leq \lfloor t/2 \rfloor$, we can repeat the above $2t+1$ times by each time replacing the honest party in S_i that get no messages with a party not in $S_i \cup \{p_1, p_2, \ldots, p_i\}$. Thus, we get that for every possible set of inputs V (values proposed by honest parties) there is a set T of $2t+1$ parties s.t. for every party $p \in T$ there is a run of A in which honest parties propose values from V, p is honest, and p does not get any messages. In particular, there exist such set T_0 for the case in which all honest party input 0 and a set T_1 for the case in which all honest parties input 1. Since $|T_0| = |T_1| = 2t + 1$, there is a party $p \in T_1 \cap T_2$. Therefore, by the Termination and Binary validity properties, there is a run r in which p does not get any messages and decides 0 and a run r' in which p does not any messages and decides 1. However, since r and r' are indistinguishable to p we get a contradiction. ◀

The following Theorem follows directly from Lemma 7 and Claim 5.

▶ **Theorem 2** (restated). *Any synchronous deterministic Byzantine agreement protocol has a communication cost of* $\Omega(ft + t)$.

4 Asymptotically optimal optimistic Byzantine Agreement

Our optimistic Byzantine agreement protocol safely combines synchronous and asynchronous protocols. Our synchronous protocol, which is interesting on its own, matches the lower bound proven in Theorem 2. That is, its communication complexity is $O(ft + t)$. The asynchronous protocol we use has a worst-case optimal quadratic communication complexity. For ease of exposition, we construct our protocol in steps. First, in Section 4.1, we present the local state each party maintains, define the *leader-based view (LBV)* [37] building block, which is used by both protocols, and present an implementation with $O(n)$ communication complexity. Then, in Section 4.2, we describe our synchronous protocol, and in Section 4.3 we use the LBV building block to reconstruct VABA [1] - an asynchronous Byzantine agreement protocol with expected $O(n^2)$ communication cost and $O(1)$ running time. Finally, in section 4.4, we safely combine both protocols to prove the following:

▶ **Theorem 4** (restated). *There is an authenticated Byzantine agreement protocol with* $O(ft + t)$ *communication complexity in synchronous runs and expected* $O(t^2)$ *communication complexity in all other runs.*

The correctness proof and communication analysis of the protocol appear in the fullpaper [36].

4.1 General structure

The protocol uses many instances of the LBV building block, each of which is parametrized with a sequence number and a leader. We denote an LBV instance that is parametrized with sequence number sq and a leader p_l as $LBV(sq, p_l)$. Each party in the protocol maintains a local state, which is used by all LBVs and is updated according to their returned values. Section 4.1.1 presents the local state and Section 4.1.2 describes a linear communication LBV implementation. Section 4.1.3 discusses the properties guaranteed by a sequential composition of several LBV instances.

4.1.1 Local state

The local state each party maintains is presented in Algorithm 1. For every possible sequence number sq, $LEADER[sq]$ stores the party that is chosen (a priori or in retrospect) to be the leader associated with sq. The $COMMIT$ variable is a tuple that consists of a value val, a sequence number sq s.t. val was committed in LBV(sq,$LEADERS$[sq]), and a threshold signature that is used as a proof of it. The $VALUE$ variable contains a safe value to propose and the KEY variable is used as proof that $VALUE$ is indeed safe. KEY contains a sequence number sq and a threshold signature that proves that no value other than $VALUE$ could be committed in LBV(sq,$LEADERS$[sq]). The $LOCK$ variable stores a sequence number sq, which is used to determine what keys are up-to-date and what are obsolete – a key is up-to-date if it contains a sequence number that is greater than or equal to $LOCK$.

Algorithm 1 Local state initialization.

```
$LOCK \in \mathbb{N} \cup \{\bot\}$, initially $\bot$
$KEY \in (\mathbb{N} \times \{0,1\}^*) \cup \{\bot\}$ with selectors $sq$ and $proof$, initially $\bot$
$VALUE \in \mathbb{V} \cup \{\bot\}$, initially $\bot$
$COMMIT \in (\mathbb{V} \times \mathbb{N} \times \{0,1\}^*) \cup \{\bot\}$ with selectors $val$, $sq$ and $proof$, initially $\bot$
for every $sq \in \mathbb{N}$, $LEADER[sq] \in \Pi \cup \{\bot\}$, initially $\bot$
```

4.1.2 Linear leader-based view

For space limitation, detailed pseudocode of the linear implementation of the LBV building block is given in the fullpaper [36]. An illustration appears in figure 1. The LBV building block supports an API to *start the view* and *wedge the view.* Upon a startView(⟨sq, p$_l$⟩) invocation, the invoking party starts processing messages associated with LBV(sq,p_l). When the leader p_l invokes startView(⟨sq, p$_l$⟩) it initiates 3 steps of leader-to-all and all-to-leader communication, named *PreKeyStep*, *KeyStep*, and *LockStep*. In each step, the leader sends its $VALUE$ together with a threshold signature that proves the safety of the value for the current step and then waits to collect $n - t$ valid replies. A party that gets a message from the leader, validates that the received value and proof are valid for the current step, then produces its signature share on a message that contains the value and the step's name, and sends the share back to the leader. When the leader gets $n - t$ valid shares, it combines them into a threshold signature and continues to the next step. After successfully generating the threshold signature at the end of the third step (*LockStep*), the leader has a commit certificate which he sends together with its $VALUE$ to all parties.

In addition to validating and share-signing messages, parties also store the values and proofs they receive. The **keyProof** and **lockProof** variables store tuples consisting of the values and the threshold signatures received from the leader in the *KeyStep*, and *LockStep* steps, respectively. The **commitProof** variable stores the received value and the commit certificate. When a party receives a valid commit certificate from the leader it returns.

As for the validation of the leader's messages, parties distinguish the *PreKeyStep* message from the rest. For *KeyStep*, *LockStep* and commit certificate messages, parties simply check that the attached proof is a valid threshold signature on the leader's value and the previous step name. The *PreKeyStep* message, however, is used by the Agreement protocols to safely compose many LBV instances. We describe this mechanism in more details below, but to develop some intuition let us first present the properties guaranteed by a single LBV instance:

- Commit causality: If a party gets a valid commit certificate, then at least $t + 1$ honest parties previously got a valid **lockProof**.
- Lock causality: If a party gets a valid **lockProof**, then at least $t + 1$ honest parties previously got a valid **keyProof**.
- Safety: All valid **keyProof**, **lockProof**, and commit certificates obtained in the same LBV have the same value.

The validation of the *PreKeyMessage* in *PreKeyStep* makes sure that the leader's value satisfies the safety properties of the Byzantine agreement protocol that sequentially composes and operates several LBVs. The *PreKeyMessage* contains the leader's $VALUE$ and KEY, where KEY stores the last (non-empty) **keyProof** returned by a previous LBV instance together with the LBV's sequence number. When a party gets a *PreKeyMessage* it first validates, by checking the key's sequence number *sq*, that the attached key was obtained in an LBV instance that does not precede the one the party is locked on (the sequence number that is stored in the party's *LOCK* variable). Then, the party checks that the threshold signature in the key (1) was generated at the end of the *PreKeyStep* step (it is a valid **keyProof**) in LBV(sq,*LEADER[sk]*); and (2) it is a valid signature on a message that contains the leader's *VALUE*. Note that if the party is not locked ($LOCK = \bot$) then a key is not required.

Upon a wedgeView(sq, $\mathsf{p_l}$) invocation, the invoking party stops participating in LBV(sq,p_l) and returns its current **keyProof**, **lockProof**, and **commitProof** values. These values are used by both synchronous and asynchronous protocols, which are built on top of LBV instances, to update the *LOCK*, *KEY*, *VALUE*, and *COMMIT* variables in parties' local states. Stopping participating in LBV(sq,p_l) upon a wedgeView(sq, $\mathsf{p_l}$) invocation guarantees that the the LBVs' causality guarantees are propagated the *KEY*, *LOCK*, and *COMMIT* variables in parties local states.

Communication complexity. Note that the number of messages sent among honest parties in an LBV instance is $O(n) = O(t)$. In addition, since signatures are not accumulated – leaders use threshold signatures – each message contains a constant number of words, and thus the total communication cost of an LBV instance is $O(t)$ words.

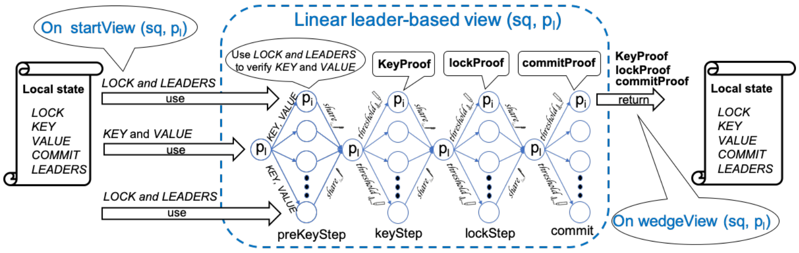


Figure 1 A linear communication LBV illustration. The local state is used by and updated after each instance. The **keyProof**, **lockProof**, and **commitProof** are returned when a commit message is received from the leader or *wedgeView* is invoked.

4.1.3 Sequential composition of LBVs

As mentioned above, our optimistic Byzantine agreement protocol is built on top of the LBV building blocks. The synchronous and the asynchronous parts of the protocol use different approaches, but they both sequentially compose LBVs - the synchronous part of the protocol determines the composition in advance, whereas the asynchronous part chooses what instances are part of the composition in retrospect.

In a nutshell, a sequential composition of LBVs operates as follows: parties start an LBV instance by invoking startView and at some later time (depends on the approach) invoke wedgeView and update their local states with the returned values. Then, they exchange messages to propagate information (e.g., up-to-date keys or commit certificates), update their local states again and start the next LBV (via startView invocation). We claim that an agreement protocol that sequentially composes LBV instances and maintains the local state in Algorithm 1 has the following properties:

- Agreement: all commit certificates in all LBV instances have the same value.
- Conditional progress: for every LBV instance, if the leader is honest, all honest parties invoke *startView*, and all messages among honest parties are delivered before some honest party invokes *wedgeView*, then all honest parties get a commit certificate.

Intuitively, by the LBV's commit causality property, if some party returns a valid commit certificate (**commitProof**) with a value v in some LBV(sq,p_i), then at least $t+1$ honest parties return a valid **lockProof** and thus lock on sq ($LOCK \leftarrow sq$). Therefore, since the leader of the next LBV needs the cooperation of $n-t$ parties to generate threshold signatures, its *PreKeyStep* message must include a valid **keyProof** that was obtained in LBV(sq,p_i). By the LBV's safety property, this **keyProof** includes the value v and thus v is the only value the leader can propose. The agreement property follows by induction.

As for conditional progress, we have to make sure that honest leaders are able to drive progress. Thus, we must ensure that all honest leaders have the most up-to-date keys. By the lock causality property, if some party gets a valid **lockProof** in some LBV, then at least $t+1$ honest parties get a valid **keyProof** in this LBV and thus are able to unlock all honest parties in the next LBV. Therefore, leaders can get the up-to-date key by querying a quorum of $n-t$ parties.

From the above, any Byzantine agreement protocol that sequentially composes LBVs satisfies Agreement. The challenge, which we address in the rest of this section, is how to sequentially compose LBVs in a way that satisfies Termination with asymptotically optimal communication complexity under all network conditions and failure scenarios.

4.2 Adaptive to failures synchronous protocol

Algorithm 2 Adaptive synchronous protocol: Procedure for a party p_i.

```
upon Synch-propose($v_i$) do
    $VALUE \leftarrow v_i$
    tryOptimistic()

procedure TRYOPTIMISTIC()
    trySynchrony($1, p_1, 7\Delta$)
    for $j \leftarrow 2$ to $n$ do
        if $i \neq j$ then
            trySynchrony($j, p_j, 9\Delta$)
        else if $COMMIT = \bot$ then
            send "KEYREQUEST" to all parties
            wait for $2\Delta$ time
            trySynchrony($j, p_j, 7\Delta$)

procedure TRYSYNCHRONY($sq, leader, T$)
    invoke startView($sq, leader$)                          ▷ non-blocking invocation
    wait for $T$ time
    $\langle keyProof, lockProof, commitProof\rangle \leftarrow$ wedgeView($sq, leader$)
    updateState($sq, leader, keyProof, lockProof, commitProof$)

upon receiving "KEYREQUEST" from party $p_k$ for the first time do
    send "KEYREPLY, $KEY$, $VALUE$" to party $p_k$

upon receiving "KEYREPLY, $key$, $value$" do
    check&updateKey(key, value)
```

In this section, we describe a synchronous Byzantine agreement protocol with an asymptotically optimal adaptive communication cost that matches the lower bound in Theorem 2. Namely, we prove the following Theorem:

▶ **Theorem 3** (restated). *There is a deterministic synchronous authenticated Byzantine agreement protocol with $O(ft + t)$ communication complexity.*

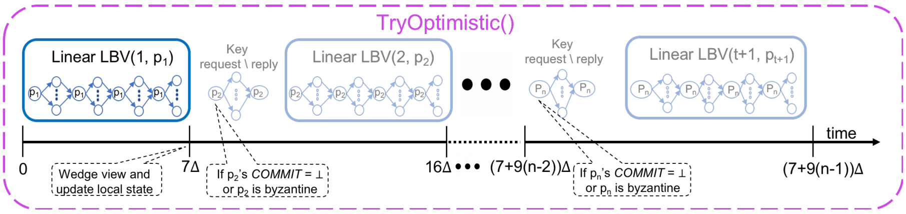

TryOptimistic()
Linear LBV(1, p_1)
p_1
Key request \ reply
p_2
Linear LBV(2, p_2)
Key request \ reply
P_n
Linear LBV(t+1, p_{t+1})
time
0
Wedge view and update local state
7Δ
If p_2's *COMMIT* = ⊥ or p_2 is byzantine
16Δ ••• (7+9(n-2))Δ
If p_n's *COMMIT* = ⊥ or p_n is byzantine
(7+9(n-1))Δ

Figure 2 Illustration of the adaptive synchronous protocol. Shaded LBVs are not executed if their leaders have previously decided.

A detailed pseudocode is given in Algorithms 2 and 3, and an illustration appears in Figure 2. The protocol sequentially composes n pre-defined LBV instances, each with a different leader, and parties decide v whenever they get a commit certificate with v in one of them. To avoid the costly view-change mechanism that is usually unavoidable in leader-based protocols, parties exploit synchrony to coordinate their actions. That is, all the startView and wedgeView invocation times are predefined, e.g., the first LBV starts at time 0 and is wedged at time 7Δ simultaneously by all honest parties. In addition, to make sure honest leaders can drive progress, each leader (except the first) learns the up-to-date key, before invoking startView, by querying all parties and waiting for a quorum of $n - t$ parties to reply.

Algorithm 3 Auxiliary procedures to update local state.

```
procedure UPDATESTATE($sq$, $leader$, $keyProof$, $lockProof$, $commitProof$)
    $LEADERS[sq] \leftarrow leader$
    if $keyProof \neq \bot$ then
        $KEY \leftarrow \langle sq, keyProof.proof \rangle$
        $VALUE \leftarrow keyProof.val$
    if $lockProof \neq \bot$ then
        $LOCK \leftarrow sq$
    if $commitProof \neq \bot$ then
        $COMMIT \leftarrow \langle commitProof.val, sq, commitProof.proof \rangle$
        decide $COMMIT.val$

procedure CHECK&UPDATEKEY($key$, $value$)
    if $(KEY = \bot \vee key.sq > KEY.sq)$ then
        if $threshold\text{-}validate(\langle$PREKEYSTEP, $key.sq$,
                    $LEADER[key.sq], value \rangle, key.proof)$ then
            $KEY \leftarrow key$
            $VALUE \leftarrow value$

procedure CHECK&UPDATECOMMIT($commit$)
    if $COMMIT = \bot$ then
        if $threshold\text{-}validate(\langle$LOCKSTEP, $commit.sq$,
          $LEADER[commit.sq], commit.val \rangle, commit.proof)$ then
            $COMMIT \leftarrow commit$
            decide $COMMIT.val$
```

Composing n LBV instances may lead in the worst case to $O(t^2)$ communication complexity – $O(t)$ for every LBV instance. Therefore, to achieve the optimal adaptive complexity, honest leaders in our protocol participate (learn the up-to-date key and invoke startView) only in case they have not yet decided. (Note that the communication cost of an LBV instance in which the leader does not invoke startView is 0 because other parties only reply to the leader's messages.) For example, if the leader of the second LBV instance is honest and has committed a value in the first instance (its $COMMIT \neq \bot$ at time 7Δ), then no message is sent among honest parties between time 7Δ and time 16Δ.

Termination and communication complexity. A naive approach to guarantee termination and avoid an infinite number of LBV instances in a leader based Byzantine agreement protocols is to perform a costly communication phase after each LBV instance. One common approach is to reliably broadcast commit certificates before halting, while a complementary one is to halt unless receiving a quorum of complaints from parties that did not decide. In both cases, the communication cost is $O(t^2)$ even in runs with one failure.

The key idea of our synchronous protocol is to exploit synchrony in order to allow honest parties to learn the decision value and at the same time help others in a small number of messages. Instead of complaining (together) after every unsuccessful LBV instance, each party has its own pre-defined time to "complain", in which it learns the up-to-date key and value and helps others decide via the LBV instance in which it acts as the leader.

By the conditional progress property and the synchrony assumption, all honest parties get a commit certificate in LBV instances with honest leaders. Therefore, the termination property is guaranteed since every honest party has its own pre-defined LBV instance, which it invokes only in case it has not yet decided. As for the protocol's total communication cost, recall that the LBV's communication cost is $O(t)$ in the worst case and 0 in case its leader already decided and thus does not participate. In addition, since all honest parties get a commit certificate in the first LBV instance with an honest leader, we get that the message cost of all later LBV instances with honest leaders is 0. Therefore, the total communication cost of the protocol is $O(ft + t)$ – at most f LBVs with Byzantine leaders and 1 LBV with an honest one.

4.3 Asynchronous fallback

In this section, we use the LBV building block to reconstruct VABA [1]. Note that achieving an optimal asynchronous protocol is not a contribution of this paper but reconstructing the VABA protocol with our LBV building block allows us to safely combine it with our adaptive synchronous protocol to achieve an optimal optimistic one. In addition, we also improve the protocol of VABA in the following ways: first, parties in VABA [1] never halt, meaning that even though they decide in expectation in a constant number of rounds, they operate an unbounded number of them. We fix it by adding an auxiliary primitive, we call *help&tryHalting* in between two consecutive waves. Second, VABA guarantees probabilistic termination in all runs, whereas our version also guarantees standard termination in eventually synchronous runs. For space limitation, the details are given in Appendix B.

4.4 Optimal optimistic protocol: combine the pieces

Algorithm 4 Optimistic byzantine agreement: protocol for a party p_i.

```
upon Optimistic-propose($v_i$) do
    $VALUE \leftarrow v_i$
    tryOptimistic()
    help&tryHalting(n)                              ▷ Blocking invocation
    fallback(n)
```

At a high level, parties first optimistically try the synchronous protocol (of section 4.2), then invoke *help&tryHalting* and continue to the asynchronous fallback (of section 4.3) in case a decision has not been reached. Pseudocode is given in Algorithm 4 and an illustration appears in Figure 3. The parameters passed in Algorithm 4 synchronize the LBV sequence numbers across the different parts of the protocol.

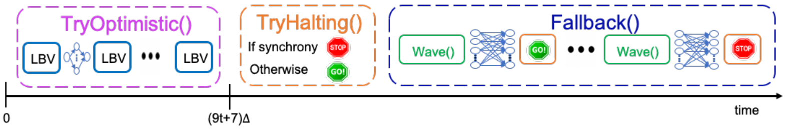


Figure 3 Illustration of the optimistic protocol. Both parts form a sequential composition of LBV instances.

One of the biggest challenges in designing an agreement protocol as a combination of other protocols is to make sure safety is preserved across them. Meaning that parties must never decide differently even if they decide in different parts of the protocol. In our protocol, however, this is inherently not a concern. Since both parts use LBV as a building block, we get safety for free. That is, if we look at an execution of our protocol in retrospect, i.e, ignore all LBVs that were not elected in the asynchronous part. Then the LBV instances in the synchronous part together with the elected ones in the asynchronous part form a sequential composition, which satisfies the Agreement property.

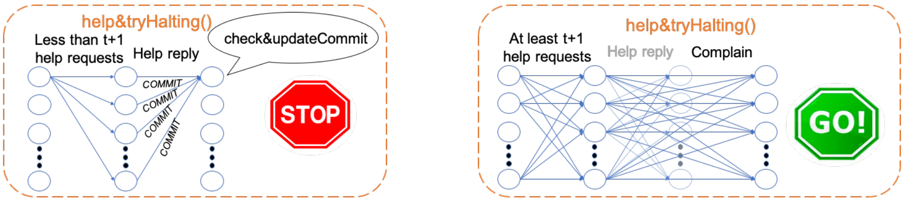
On the other hand, satisfying termination without sacrificing optimal adaptive complexity is a non-trivial challenge. Parties start the protocol by optimistically trying the synchronous part, but unfortunately, at the end of the synchronous part they cannot distinguish between the case in which the communication was indeed synchronous and all honest parties decided and the case in which some honest parties did not decide due to asynchrony. Moreover, honest parties cannot distinguish between honest parties that did not decide and thus wish to continue to the asynchronous fallback part and Byzantine parties that want to move to the fallback part to increase the communication cost.

To this end, we implement the *help&tryHalting* procedure, which stops honest parties from moving to the fallback part in synchronous runs. The communication cost of *help&tryHalting* is $O(ft)$. The idea is to help parties learn the decision value and move to the fallback part only when the number of help request indicates that the run is asynchronous.

The pseudocode of *help&tryHalting* is given in Appendix D and an illustration appears in Figure 4. Each honest party that has not yet decided sends a share signed HELPREQUEST to all other parties. When an honest party gets an HELPREQUEST, it replies with its *COMMIT* value. But if it gets $t + 1$ HELPREQUEST messages, the party combines the shares to a threshold signature and sends it in a COMPLAIN message to all. When an honest party gets a COMPLAIN message for the first time, it echos the message to all parties and continues to the fallback part. A termination intuition and complexity analysis of our full protocol are given in Appendix C.

5 Discussion and Future Directions

In this paper, we propose a new approach to design agreement algorithms for communication efficient SMR systems. Instead of designing deterministic protocols for the eventually synchronous model, which we prove cannot guarantee bounded communication cost before GST, we propose to design protocols that are optimized for the synchronous case but also have a randomized fallback to deal with asynchrony. Traditionally, most SMR solutions



(a) A few HELPREQUEST messages – help and halt.

(b) Too much HELPREQUEST messages – the run is asynchronous, move to the fallback part.

Figure 4 An illustration of the *help&tryHalting* procedure.

avoid randomized asynchronous protocols due to their high communication cost. We, in contrast, argue that this communication cost is reasonable given that the alternative is an unbounded communication cost during the wait for eventual synchrony.

We present the first authenticated optimistic protocol with $O(ft + t)$ communication complexity in synchronous runs and $O(t^2)$, in expectation, in non-synchronous runs. To strengthen our result, we prove that no deterministic protocol (even if equipped with perfect cryptographic schemes) can do better in synchronous runs. As for the asynchronous runs, the lower bound in[1] proves that $O(t^2)$ is optimal in the worst case of $f = t$.

Future work. Note that our synchronous protocol satisfies early decision but not early stopping. That is, all honest parties decide after $O(f)$ rounds, but they terminate after $O(t)$. Therefore, a natural question to ask is whether exist an early stooping synchronous Byzantine agreement protocol with an optimal adaptive communication cost. In addition, it may be possible to improve our protocol's complexity even further. In particular, the lower bound on communication cost in synchronous runs applies only to deterministic algorithms, so it might be possible to circumvent it via randomization [8].

Another interesting future direction is the question of optimal resilience in synchronous networks. Due to the lower bound in [5], the resilience of our protocol is optimal since the resilience in synchronous runs cannot be improved as long as the resilience in asynchronous runs is the optimal $t < n/3$. However, if we consider synchronous networks in which we do not need to worry about asynchronous runs, we know that we can tolerate up to $t < n/2$ failures. The open question is therefore the following: is there a synchronous Byzantine agreement protocol that tolerates up to $t < n/2$ failures with an optimal communication complexity of $O(ft + t)$?

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *PODC*. ACM, 2019.

2 Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In *IWDA*, 1996.

3 Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.

4 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In *Theory of Cryptography Conference*, 2020.

5 Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*. Springer, 2019.

6 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

7 Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*, 2001.

8 Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast byzantine agreement. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, 2013.

9 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *DISC*. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik, 2020.

10 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint*, 2018. `arXiv:1807.04938`.

11 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology*, 2001.

12 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

13 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *DISC 2020*, 2020.

14 Danny Dolev and Rudiger Reischuk. Bounds on information exchange for byzantine agreement. *JACM*, 1985.

15 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM (JACM)*, 1990.

16 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 1985.

18 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, 2010.

19 Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. Speculative linearizability. *ACM Sigplan Notices*, 47(6):55–66, 2012.

20 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: a scalable and decentralized trust infrastructure. In *DSN 2019*. IEEE, 2019.

21 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 2003.

22 Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *Journal of the ACM (JACM)*, 63(2):13, 2016.

23 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, 2007.

24 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In *OPODIS*, 2019.

25 Dahlia Malkhi. *Concurrency: The Works of Leslie Lamport.* Morgan & Claypool, 2019.

26 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *CCS*, 2019.

27 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

28 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS 2016*. ACM, 2016.

29 Atsuki Momose and Ling Ren. Optimal communication complexity of byzantine consensus under honest majority. *DISC*, 2020.

30 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *CryptoEconSys 2020*, 2019.

31 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *DISC*, 2020.

32 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. *DISC*, 2020.

33 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2), 1980.

34 Victor Shoup. Practical threshold signatures. In *ICTACT*, 2000.

35 Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

36 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. *CoRR*, abs/2002.06993, 2020. `arXiv:2002.06993`.

37 Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication. In *OPODIS' 2020*, 2020.

38 Maofan Yin, Dahlia Malkhi, MK Reiterand, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC*, 2019.

A Lower Bound For Eventually Synchronous Runs

▶ **Theorem 1** (restated). *There is no eventually synchronous deterministic Byzantine agreement protocol that can tolerate a single failure and guarantee bounded communication cost even in failure-free runs.*

Proof. Assume by a way of contradiction that there are such algorithms. Let A be such an algorithm with the lowest eventually synchronous communication cost with 0 failures, and denote its communication cost by N. Clearly, $N \geq 1$. Let $R_N \subset R_0^{es}$ be the set of all failure-free eventually synchronous runs of A that have communication cost of N. For every run $r \in R_N$ let m_r be the last message that is delivered in r, let t_r be the time at which it is delivered, and let p_r be the party that sends m_r. Now for every $r \in R_N$ consider a run r' that is identical to r up to time t_r except p_r is Byzantine that acts exactly as in r but does not send m_r. Denote by R_{N-1} the set of all such runs and consider two cases:

- There is a run $r' \in R_{N-1}$ in which some message m by an honest party p is sent at some time $t_{r'} > t_r$. Now consider a failure-free run r'' that is identical to run r except the delivery of m_r is delayed to $t_{r'} + 1$. The runs r'' and r' are indistinguishable to all parties that are honest in r' and thus p sends m at time time $t_{r'} > t_r$ in r'' as well. Therefore, the communication cost of r'' is at least $N + 1$. A contradiction to the communication cost of A.
- Otherwise, we can construct an algorithm A' with a better eventually synchronous communication cost with 0 failures than A in the following way: A' operates identically to A in all runs not in R_N and for every run $r \in R_N$ A' operates as A except p_r does not send m_r. A contradiction to the definition of A. ◀

B Fallback Description

On a high level, the idea in VABA [1] that was later generalized in ACE [37] is the following: instead of having a pre-defined leader in every "round" of the protocol as most eventually synchronous protocols have, they let n leaders operate simultaneously and then randomly choose one in retrospect. This mechanism is implemented inside a wave and the agreement protocol operates in a *wave*-by-*wave* manner s.t. parties exchange their local states between every two conductive waves. To ensure halting, in our version of the protocol, parties also invoke the *help&tryHalting* procedure after each wave. See Figure 5 for an illustration. A full detailed pseudocode of our fallback protocol can be found in the fullpaper [36].

Wave-by-wave approach. To implement the wave mechanism we use our LBV and two auxiliary primitives: Leader-election and Barrier-synchronization. At the beginning of every wave, parties invoke, via startView, n different LBV instances, each with a different leader.

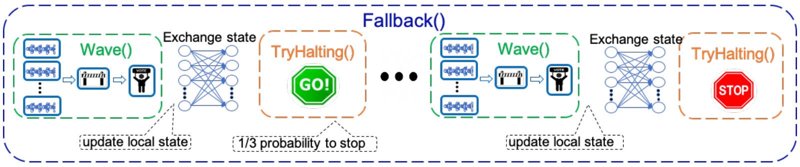


Figure 5 Asynchronous fallback. Usig linear LBV to reconstruct the VABA [1] protocol.

Then, parties are blocked in the Barrier-synchronization primitive until at least $n - 2t$ LBV instances *complete*. (An LBV completes when $t + 1$ honest parties get a commit certificate.) Finally, parties use the Leader-election primitive to elect a unique LBV instance, wedge it (via wedgeView), and ignore the rest. With a probability of 1/3 parties choose a completed LBV, which guarantees that after the state exchange phase all honest parties get a commit certificate, decide, and halt in the *help&tryHalting* procedure. Otherwise, parties update their local state and continue to the next wave. An illustration appears in figure 6.

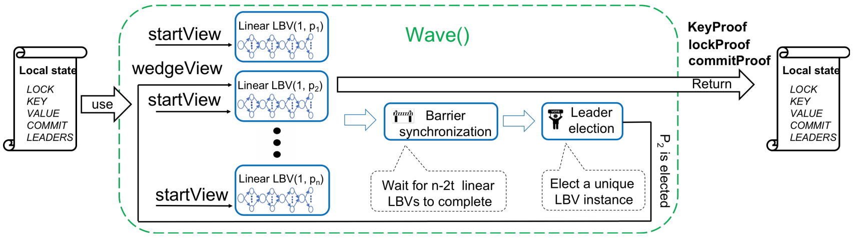


Figure 6 An illustration of a single wave. The returned **keyProof**, **lockProof**, and **commitProof** are taken from the elected LBV.

Since every wave has a probability of 1/3 to choose a completed LBV instance, the protocol guarantees probabilistic termination – in expectation, all honest parties decide after 3 waves. To also satisfy standard termination in eventually synchronous runs, we "try synchrony" after each unsuccessful wave. Between every two conjunctive waves parties deterministically try to commit a value in a pre-defined LBV instance. The preceding *help&tryHalting* procedure guarantees that after GST all honest parties invoke startView in the pre-defined LBV instance with at most 1Δ from each other and thus setting a timeout to 8Δ is enough for an honest leader to drive progress. Description of the Barrier-synchronization and Leader-election primitives can be found in [37].

C Protocol Termination Intuition And Complexity Analysis

Termination. A formal proof of Safety and Liveness is given in the fullpaper [36]. Here we provide some intuition. Consider two cases. First, the parties move to the fallback part, in which case (standard) termination is guaranteed in eventually synchronous runs and probabilistic termination is guaranteed in asynchronous runs. Otherwise, less than $t + 1$ parties send HELPREQUEST in *help&tryHalting*, which implies that at least $t+1$ honest parties decided and had a commit certificate before invoking *help&tryHalting*. Therefore, all honest parties that did not decide before invoking *help&tryHalting* eventually get a HELPREPLY message with a commit certificate and decide as well.

Note that termination does not mean halting. In asynchronous runs, HELPREQUEST messages may be arbitrary delayed and thus parties cannot halt the protocol after deciding in the synchronous part. However, it is well known and straightforward to prove that halting cannot be achieved with $o(t^2)$ communication cost in asynchronous runs, and thus our protocol is optimal in this aspect.

Round complexity. Since in synchronous runs all parties decide at the end of an LBV instances with an honest leader, we get that the round complexity in synchronous runs is $O(f+1)$. Since in asynchronous runs parties may go though n LBV instances without deciding before starting the fallback, we get that the round complexity in asynchronous runs is $O(n+1)$ in expectations.

Communuication complexity. The synchronous (optimistic) part guarantees that if the run is indeed synchronous, then all honest parties decide before invoking *help&tryHalting*. The *help&tryHalting* procedure guarantees that parties continue to the fallback part only if $t+1$ parties send an helpRequest message, which implies that they move only if at least one honest party has not decided in the synchronous part. Therefore, together they guarantee that honest parties never move to the fallback part in synchronous runs.

The communication complexity of the synchronous part is $O(ft+t)$, so to show that the total communication cost of the protocol in synchronous runs is $O(ft+t)$ we need to show that the cost of *help&tryHalting* is $O(ft+t)$ as well. Since in synchronous runs all honest parties decide in the synchronous part, they do not send helpRequest messages, and thus no party can send a valid complain message. Each Byzantine party that does send helpRequest messages can cause honest parties to send $O(t)$ replies, which implies a total communication cost of $O(ft)$ in synchronous runs.

As for all other runs, Theorem 1 states that deterministic protocols have an unbounded communication cost in the worst case. Thanks to the randomized fallback, our protocol has a communication cost of $O(t^2)$ in expectation.

D Help&tryHalting Pseudocode

Algorithm 5 Help and try halting: Procedure for a party p_i.

```
Local variables initialization:
    $S_{help} = \{\}$; $HALT \leftarrow true$

procedure help&tryHalting($sq$)
    if $COMMIT = \bot$ then
        $\rho \leftarrow share\text{-}sign_i(\langle$helpRequest$, sq\rangle)$
        send "helpRequest, $sq, \rho$" to all parties
    wait until $HALT = false$

upon receiving "helpReply, $sq, commit$" do
    check&updateCommit($commit$)

upon receiving "helpRequest, $sq, \rho$" from a party $p_j$ do
    if $share\text{-}validate(\langle$helpRequest$, sq\rangle, p_j, \rho)$ then
        $S_{help} \leftarrow S_{help} \cup \{\rho\}$
        send "helpReply, $sq, COMMIT$" to $p_j$
        if $|S_{help}| = t+1$ then
            $\nu \leftarrow threshold\text{-}sign(S_{help})$
            send "complain, $sq, \nu$" to all parties

upon receiving "complain, $sq, \nu$" do
    if $threshold\text{-}validate(\langle$helpRequst$, sq\rangle, \nu)$ then
        send "complain, $sq, \nu$" to all parties
        $HALT \leftarrow false$
```

The Power of Random Symmetry-Breaking in Nakamoto Consensus

Lili Su
Northeastern University, Boston, MA, USA

Quanquan C. Liu
Massachusetts Institute of Technology, Cambridge, MA, USA

Neha Narula
Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

Nakamoto consensus underlies the security of many of the world's largest cryptocurrencies, such as Bitcoin and Ethereum. Common lore is that Nakamoto consensus only achieves consistency and liveness under a regime where the difficulty of its underlying mining puzzle is very high, negatively impacting overall throughput and latency. In this work, we study Nakamoto consensus under a wide range of puzzle difficulties, including very easy puzzles. We first analyze an adversary-free setting and show that, surprisingly, the common prefix of the blockchain grows quickly even with easy puzzles. In a setting with adversaries, we provide a small backwards-compatible change to Nakamoto consensus to achieve consistency and liveness with easy puzzles. Our insight relies on a careful choice of *symmetry-breaking strategy*, which was significantly underestimated in prior work. We introduce a new method – *coalescing random walks* – to analyzing the correctness of Nakamoto consensus under the uniformly-at-random symmetry-breaking strategy. This method is more powerful than existing analysis methods that focus on bounding the number of *convergence opportunities*.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Nakamoto consensus, Byzantine consensus, blockchain, symmetry-breaking, coalescing random walks

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.39

Related Version *Full Version*: https://arxiv.org/pdf/2108.09604.pdf

1 Introduction

Nakamoto consensus [19], the elegant blockchain protocol that underpins many cryptocurrencies, achieves consensus in a setting where nodes can join and leave the system without getting permission from a centralized authority. Instead of depending on the identity of nodes, it achieves consensus by incorporating computational puzzles called proof-of-work [9] (also known as *mining*) and using a simple longest-chain protocol.[1] Nodes in a network maintain a local copy of an append-only ledger and gossip messages to add to the ledger, collecting many into a block. A block consists of the set of records to add, a pointer to the previous block in the node's local copy of the ledger, and a nonce, which is evidence the node has done proof-of-work, or solved a computational puzzle of sufficient difficulty, dependent on the block. The node then broadcasts its local chain to the network. Honest nodes choose a chain they see with the most proof-of-work to continue building upon.

Previous work defined correctness and liveness in proof-of-work protocols (also referred to as the *Bitcoin backbone*) using three properties: *common-prefix*, *chain-quality*, and *chain-growth* [12, 15, 21]. Informally, common-prefix indicates that any two honest nodes share a

[1] We use "longest chain" to mean the one with the most proof-of-work given difficulty adjustments, not necessarily the one with the most blocks, though without considering difficulty adjustments they are the same.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 39; pp. 39:1–39:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

common prefix of blocks, chain-growth is the rate at which the common prefix grows over time, and chain-quality represents the fraction of blocks created by honest nodes in a chain. In previous work, achieving these properties critically relied on the setting of the difficulty factor in the computational puzzles. We express this as p, the probability that any node will solve the puzzle in a given round. Previous work analyzing Nakamoto consensus has shown that for consistency and liveness p should be very small in relation to the expected network delay and the number of nodes [12, 21]. For example, mining difficulty in Bitcoin is set so that the network is only expected to find a puzzle solution roughly once every ten minutes.

Requiring a small p increases block time, removing a parameter for improving transaction throughput. One way to compensate is by increasing block size, which could result in burstier network traffic and longer transaction confirmation times for users. Newer chains which do not use proof-of-work seem to favor short block times, probably because users value a fast first block confirmation: in EOS, blocks are proposed every 500 milliseconds [10] and Algorand aims to achieve block finality in 2.5 seconds [18], whereas in Bitcoin blocks only come out every ten minutes.

Common belief is that larger p fundamentally constrains chain growth (i.e., the growth of the common prefix), even in the absence of an adversary, due to the potential of increased *forking*: nodes will find puzzle solutions (and thus blocks) at the same time; because of the delay in hearing about other nodes' chains nodes will build on different chains, delaying agreement. Another common conjecture, explicitly mentioned in [12], is that the choice of *symmetry-breaking strategies*, or ways honest nodes choose among multiple longest chains, is not relevant to correctness.

In this paper, we show that these common beliefs are incorrect. In particular, we show that when p is beyond the well-studied region even the simple strategy of choosing among chains of equal length randomly fosters chain growth, especially in the absence of adversaries.

Contributions. In this work, we formally analyze Nakamoto consensus under a wide range of p including large p. We confirm previous (informal) analysis that Nakamoto consensus requires small p in the presence of adversaries, but show that surprisingly, it does not in a setting without adversaries, even if $p = 1$ (all nodes mine blocks every round) with a minor change in nodes' symmetry-breaking strategy. Previous work assumed the requirement of *convergence opportunities*, a period when only one honest node mines a block, in order to achieve consistency [17, 21]; we show that in fact convergence opportunities are *not* required for common-prefix and chain growth. With an additional backwards-compatible modification to Nakamoto consensus, we can derive a bound on the chain growth for a wider range of p (including large p) in a setting with adversaries. Our key idea in this modification is to introduce a *verifiable delay function* [5] to prevent the adversaries from extending a chain by multiple blocks in a round. Our analysis is based on a new application of a well-known technique, coalescing random walks. To our knowledge this is the first application of coalescing random walks to analyze the common-prefix and chain quality of Bitcoin and other proof-of-work protocols. We thoroughly analyze Nakamoto consensus with the *uniformly-at-random* symmetry-breaking strategy and discuss different symmetry-breaking strategies including *first-seen*, *lexicographically-first*, and *global-random-coin*.

In summary, our contributions are as follows:

- A new approach for analyzing the confirmation time of the Bitcoin protocol under the uniformly-at-random symmetry-breaking strategy in the adversarial-free setting via *coalescing random walks*. Our analysis works for a new region of p, and shows that previous works' requirement for *convergence opportunities* was unneeded.

- New notions of *adversarial advantages* and *coalescing opportunities* to provide a more general analysis of common-prefix and chain growth in Nakamoto consensus in the presence of adversaries.

Related Work. Proofs-of-work were first put forth by Dwork and Naor [9]. Garay, Kiayias, and Leonardas [12] provided the first thorough analysis of Nakamoto's protocol in a synchronous static setting, introducing the ideas of *common-prefix*, *chain quality* and *chain growth*. Later work [15] extended the analysis to a variable difficulty function. Pass, Seeman, and shelat [21] extended the idea of common-prefix to *future self-consistency*, and provided an analysis of Nakamoto consensus in the semi-synchronous setting with an adaptive adversary. Several additional papers used this notion of future self-consistency [17,27]. [17,21] relied on *convergence opportunities*, or rounds where only one node mines a block, to analyze chain growth. In this work we show that convergence opportunities are *not* required for chain growth, and relying on them underestimates chain growth with high p; in the adversary-free setting we show chain growth even with $p = 1$ (no convergence opportunities; all nodes mine a block every round). Other work considered the tradeoffs between chain growth and chain quality [15,16,21,23,26]; however, to the best of our knowledge, none of these works considered different symmetry breaking strategies to enable faster chain growth while maintaining chain quality. In our paper, we thoroughly explore this domain. Another line of work [11,25] considers how the uniformly-at-random symmetry breaking strategy affects incentive-compatible selfish mining attacks; our analysis applies to general attacks.

Random walks have been used to analyze the probability of consistency violations in proofs-of-stake protocols [3]; ours is the first work that uses coalescing random walks to analyze the common-prefix and chain quality of Bitcoin and other proof-of-work protocols.

2 Model and Definitions

In this section, we present the specific model we use and briefly describe the Bitcoin cryptosystem. We follow the formalization presented in [15,17,21].

Network and Computation Model. Following previous work [12,14,15,21,24,27], we consider a synchronous network where nodes send messages in synchronous rounds, i.e., $\Delta = 1$; equivalently, there is a global clock and the time is slotted into equal duration rounds. Each node has identical computing power. Notably, the synchronous rounds assumption is significantly more relaxed than assuming $\Delta = 0$.[2] Our model operates in the *permissionless setting*. This means that any miner can join (or leave) the protocol execution without getting permission from a centralized or distributed authority. For ease of exposition, we assume the number of participants remains n. Our results can be easily generalized to handle perturbation in the population size by a stochastic dominance argument as long as the population size does not deviate too far from n, and the proportion of Byzantine participants does not increase due to the perturbation.

Adversary Model. Throughout this paper, we assume that all Byzantine nodes are controlled by a *probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ that can coordinate the behavior of all such nodes. $\mathcal{A}$ operates in PPT which means they have access to random coins but can only

[2] In fact, the analysis based on Poisson race [2,20] essentially assumes all mined blocks can be ordered in a globally consistent way, i.e., $\Delta = 0$, which does not hold in our synchronous network model.

use polynomial time to perform computations. At any time during the run of the protocol, $\mathcal{A}$ can corrupt up to b nodes at any point in time where b is a parameter that is an input to the protocol. The corrupted nodes remain corrupted for the remainder of the protocol. Finally, $\mathcal{A}$ cannot modify or delete the messages sent by honest nodes, but can read all messages sent over the network and arbitrarily order the messages received by any honest nodes.

2.1 Bitcoin Cryptosystem

A *blockchain protocol* is a stateful algorithm wherein each node maintains a local version of the blockchain $\mathcal{C}$. Each honest node runs its own homogeneous version of the blockchain protocol. Nodes receive messages from the *environment* $\mathcal{Z}(1^\lambda)$, where λ is the security parameter chosen based on the population size n. The environment is responsible for all the external factors related to a protocol's execution. For example, it provides the value of b to the nodes. Detailed description of the environment can be found in [21].

The protocol begins by having the environment $\mathcal{Z}$ initialize n nodes. The protocol proceeds in synchronous rounds; at each round r, each node receives a message from $\mathcal{Z}$. In each round, an honest node attempts to mine a block containing its message to add to its local chain. We provide formal definitions of the Bitcoin cryptosystem below.

Blocks and Blockchains. A blockchain $\mathcal{C} \triangleq B_0B_1B_2\cdots B_\ell$ for some $\ell \in \mathbb{N}$ is a chain of blocks. Here B_0 is a predetermined *genesis block* that all chains must build from. A *block* B_ℓ, for $\ell \geq 1$, is a triple $B_\ell = \langle s, x, \mathsf{nce}\rangle$, where $s, x, \mathsf{nce} \in \{0,1\}^*$ are three binary strings of arbitrary length. Specifically, s is used to indicate this block's predecessor, x is the text of the block containing the message (e.g. transactions) and other metadata, and nce is a *nonce* chosen by a node.

Proofs-of-Work. The Bitcoin cryptosystem crucially uses nonces as *proofs-of-work* for determining whether a block can be legally added to a chain.[3] Proof-of-work (PoW) is rigorously defined in previous work [12, 14, 15, 21, 24, 27] based on the use of the *random oracle model.*

▶ **Definition 1** (Random Oracle Model). *A random oracle $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$ on input $x \in \{0,1\}^*$ outputs a value selected uniformly at random from $\{0,1\}^\lambda$ if x has never been queried before. Otherwise, it returns the previous value returned when x was queried last.*

▶ **Definition 2** (Bitcoin PoW). *All nodes access a common random oracle $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$. We say a node successfully performs a PoW with* proof *$x \in \{0,1\}^*$ if $\mathcal{H}(x) \leq D$.*

▶ **Definition 3** (Valid Chain). *A blockchain $\mathcal{C} = B_0B_1\cdots B_\ell = B_0\langle s_1, x_1, \mathsf{nce}_1\rangle \cdots \langle s_\ell, x_\ell, \mathsf{nce}_\ell\rangle$ is* valid *with respect to a given puzzle difficulty level $D \in \{1, \cdots, 2^\lambda\}$ if the following hold: (1) $\mathcal{H}(B_0) = s_1$ and $\mathcal{H}(B_{\ell'}) = s_{\ell'+1}$ for $\ell' = 1, \cdots, \ell - 1$; and (2) $\mathcal{H}(B_{\ell'}) \leq D$ for $\ell' = 0, \cdots, \ell$.*

Longest Chain Rule. The length of a valid chain C is the number of blocks it contains. We refer to the local version of the blockchain kept by node i as the local chain at node i, denoted by $\mathcal{C}_i$. In each round r, node i tries to mine a block via solving a PoW puzzle with

[3] Note that in practice, the nonce is effectively concatenated with a miner's public key (included in the *coinbase* transaction) to ensure unique queries. The public key does *not* need to be verified. Importantly, this means that the miner can just generate a (pk, sk) pair on their local computer without the need to verify that identity with a third-party authority.

the specified difficulty D. If a block is successfully mined, then node i extends its local chain with this block and broadcasts its updated local chain to all other nodes in the network, which will be delivered at each node at the beginning of the next round. At the beginning of the next round, before working on PoW, node i updates its local chain to be the longest chain it has seen. If there are many longest chains, node i chooses one of them uniformly at random.

For ease of exposition, henceforth, C_i is referred to the local chain at the end of a round; $C_i(t)$ is the local chain of node i at the end of round t. Equivalent to using the difficulty parameter D, one can instead consider $p \triangleq D/2^{\lambda}$. The notion of p used in lieu of D has been considered in [12, 14, 15, 17, 21, 24] to simplify notation. Henceforth, we will quantify the algorithm performance in terms of p rather than D and λ.

We use the phrase *with overwhelming probability* throughout this paper. *With overwhelming probability* is defined as with probability at least $1 - \frac{1}{\text{poly}(\lambda)^c}$ for any constant $c \geq 1$. We use the phrase *with all but negligible probability in* λ to mean that the probability is upper bounded by some negligible function $\nu(\lambda)$ on λ (defined in Definition 4).

▶ **Definition 4** (Negligible Probability). *A function ν is* negligible *if for every polynomial $p(\cdot)$, there exists an N such that for all integers $n > N$, it holds that $\nu(n) < \frac{1}{p(n)}$. We denote such a function by* negl*. An event that occurs with* negligible probability *occurs with probability* negl(n)*.*

2.1.1 Properties of the Protocol

In this paper, we will analyze the Nakamoto consensus in terms of two characteristics (generalized from definitions in [12, 17, 27]). The *common prefix* is defined as a sub-chain that is a common prefix of the local chains of all honest nodes at the end of a round. The two properties *maximal common prefix* and *maximal inconsistency* are defined intuitively as: the maximal prefix that is the same across all honest chains and the maximal number of blocks in any honest chain that is not shared by all other honest chains, respectively.

▶ **Property 5** (Maximal common-prefix and maximal inconsistency). *Given a collection of chains $\mathcal{C} = \left\{\tilde{C}_1, \cdots, \tilde{C}_m\right\}$ that are kept by honest nodes, the maximal common-prefix of chain set $\mathcal{C}$, denoted by $P_{\mathcal{C}}$, is defined as the longest common-prefix of chains $\tilde{C}_1, \cdots, \tilde{C}_m$. The maximal inconsistency of $\mathcal{C}$, denoted by $I_{\mathcal{C}}$, is defined as*

$$\max_{i:1\leq i\leq m} \left|\tilde{C}_i - P_{\mathcal{C}}\right|, \tag{1}$$

where $\tilde{C}_i - P_{\mathcal{C}}$ is the sub-chain of $\tilde{C}_i$ after removing the prefix $P_{\mathcal{C}}$ and $|\cdot|$ denotes the length of the chain, i.e., the number of blocks in the chain.

3 Fundamental Limitations of Existing Approaches

To the best of our knowledge, existing work assumes extremely small p. In fact, the seemingly mild *honest majority assumption* in [13, 22] also implicitly assumes small p.

▶ **Proposition 6.** *If the* honest majority assumption *in [13] holds, then $p \leq \frac{n-2b}{2(n-b)^2}$.*

A formal statement of the honest majority assumption and the proof of Proposition 6 can be found in the full version. Note that the upper bound in this proposition is only a necessary condition. Having p satisfy this condition does not guarantee protocol correctness.

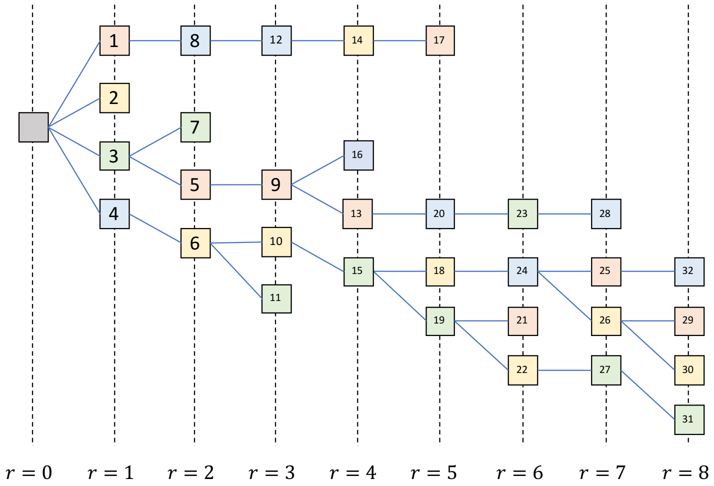


Figure 1 Example growth of a set of chains starting with the genesis block at round $r = 0$. Here, in this example $p = 1$, $n = 4$, and $b = 0$.

▶ Remark 7. Proposition 6 implies that in the vanilla Nakamoto consensus protocol, unless $\frac{b}{n}$ is *non-trivially* bounded above from $\frac{1}{2}$, p needs to be extremely low – even much lower than the commonly believed $\Theta(\frac{1}{n})$. See the full version. for detailed arguments.

To the best of our knowledge, most of the existing analyses focus on bounding the number of "convergence opportunities", which for $\Delta = 1$ is defined as the number of rounds in which *exactly* one honest node mines a block, and for general Δ, it is defined as the global block mining pattern that consists of (i) a period of Δ rounds where no honest node mines a block, (ii) followed by a round where a single honest player mines a block, (iii) and, finally, another Δ rounds of silence from the honest nodes [17,21]. Obviously, guaranteeing sufficiently many convergence opportunities necessarily requires p to be small; in the extreme case when $p = 1$ there will be no convergence opportunities at all. An important insight from our results is that *convergence opportunities are not necessary for common-prefix growth.* This is illustrated Fig. 1 which depicts the chain growth when there are 4 honest nodes and $p = 1$. Each node mines a block every round and each is associated with a color. In particular, blocks $1, 5, 9, 13, 17, 21, 25, 29$ are mined by the pink node, blocks $4, 8, 12, 16, 20, 24, 28, 32$ are mined by the blue node, etc. In each round, each node chooses one of the existing longest chains uniformly at random to extend. As shown in Fig. 1, there are no convergence opportunities in any of these 8 rounds and the four nodes never choose the same chain to extend. However, instead of the trivial common prefix (the genesis block) the longest chains at the end of round 8 (the four chains ending with blocks 32, 29, 30, and 31, respectively) share the common prefix *genesis* $\to 4 \to 6 \to 10 \to 15$. In general, as we show in Section 4, even for the extreme case when $p = 1$, the common prefix of the longest chains still grows as time goes by.

4 Uniformly-at-Random Symmetry-Breaking Strategy

Bitcoin uses the *first-seen* symmetry-breaking strategy; nodes will only switch to a new chain with more proof-of-work than their current longest chain. In this section, we investigate the power of the uniformly-at-random symmetry-breaking strategy, in which each honest node chooses one of its received longest chains uniformly at random to extend upon – independently of other nodes and independently across rounds. We choose to start with the uniformly-at-random strategy because (1) it is easy to implement, especially in a distributed fashion, and (2) despite its simplicity, it is very powerful in fostering chain growth.

For ease of exposition, we first present our results in the adversary-free setting (Sections 4.1 and 4.2) and then in the adversary-prone setting (Section 4.3).

4.1 Warmup: $p = 1$ and Adversary-Free

Even the adversary-free setting (i.e., $b = 0$) is surprisingly non-trivial to analyze. Hence we build insights by first considering the simpler setting where $p = 1$ as a warmup.

▶ **Theorem 8.** *Suppose that $p = 1$ and $b = 0$. Then for any given round index $t \geq 1$, in expectation, the local chains at the honest nodes share a common prefix of length $t + 1 - O(n)$.*

▶ Remark 9. In Theorem 8, the expectation is taken w. r. t. the randomness in the symmetry breaking strategy. Theorem 8 says that large p indeed boosts the growth of the common prefix among the local chains kept by the honest nodes, and that, though temporal forking exists among local chains kept by the honest nodes, such forking can be quickly resolved by repetitive symmetry-breaking across rounds.

The following definition and theorem are useful to see the intuitions of Theorem 8.

▶ **Definition 10** (Coalescing Random Walks [1][4])**.** *In a coalescing random walk, a set of particles make independent random walks on a undirected graph $G = (V, E)$ with self-loops. Whenever one or more particles meet at a vertex, they unite to form a single particle, which then continues the random walk through the graph. We define the* coalescence time*, denoted by C_G, to be the number of steps required before all particles merge into one particle.*

▶ **Theorem 11** ([1, 7])**.** *If $G = (V, E)$ is complete, then $\mathbb{E}[C_G] = O(n)$.*

In the proof of Theorem 8, we build up the connection between the longest chains and the backwards coalescing random walks on complete graphs, and show that the maximal inconsistency among n longest chains turns out to be the same as the number of steps it takes n random walks on the n-complete graph to coalesce into one. Finally, we use the existing results on coalescing random walks to conclude.

Main proof ideas of Theorem 8. We cast our proof insights via an example presented in Fig. 1. In this figure, there are four miners. For ease of exposition, we use the colors *pink*, *yellow*, *green*, and *blue* to represent each of the miners, respectively. As shown in Fig. 1, there are 4 longest chains at the end of round 8 and these chains share a maximal common prefix ending at block 15. The maximal inconsistency of these 4 longest chains is 4; that is, these 4 longest chains are NOT inconsistent with each other until the most recent 4 blocks of each chain. For expository convenience below, instead of using numbers to represent each of the blocks, we use the tuple (color, r) to represent a block that is mined by a certain miner at round r. The maximal inconsistency of the longest chains can be characterized by the coalescing time on complete graphs. To see this, let's consider the four longest chains held by honest miners during round 8 backwards.

Backwards-Chain #1: (**blue**, 8) → (pink, 7) → (**blue**, 6) → (yellow, 5) → (**green**, 4) → (yellow, 3) → (yellow, 2) → (**blue**, 1) → (gray, 0), which can be read as "block (**blue**, 8) is attached to block (pink, 7) which is further attached to block (**blue**, 6) ... attached to the genesis block (gray, 0). "

[4] The original definition given in [1] assumes no self-loops, but its analysis applies to the graphs with self-loops.

Backwards-Chain #2: $(\text{pink}, 8) \to (\text{yellow}, 7) \to (\text{pink}, 6) \to (\textbf{green}, 5) \to (\textbf{green}, 4) \to (\text{yellow}, 3) \to (\text{yellow}, 2) \to (\textbf{blue}, 1) \to (\text{gray}, 0)$.
Backwards-Chain #3: $(\text{yellow}, 8) \to (\text{yellow}, 7) \to (\text{pink}, 6) \to (\textbf{green}, 5) \to (\textbf{green}, 4) \to (\text{yellow}, 3) \to (\text{yellow}, 2) \to (\textbf{blue}, 1) \to (\text{gray}, 0)$.
Backwards-Chain #4: $(\textbf{green}, 8) \to (\textbf{green}, 7) \to (\text{yellow}, 6) \to (\textbf{green}, 5) \to (\textbf{green}, 4) \to (\text{yellow}, 3) \to (\text{yellow}, 2) \to (\textbf{blue}, 1) \to (\text{gray}, 0)$.

Since $p = 1$ and there is no adversary, the number of longest chains received by each honest node at each round is n. Under our symmetry-breaking rule, in each round t, each miner chooses which of the longest chains received at the beginning of round t to extend on uniformly-at-random. Thus, neither the previous history up to round t nor the future block attachment choices after round t affects the choice of the chain extension in round t. Reasoning heuristically[5], we can view each of the backwards-chain as a random walk on a 4-complete graph with vertex set $\{pink, yellow, green, blue\}$. In particular, Backwards-Chain #1 can be viewed as a sample path of a random walk starting at the blue vertex, then moves to the pink vertex, then back to the blue vertex etc., and finally to the blue vertex. Similarly, Backwards-Chains #2, #3, and #4 can be viewed as the sample paths of three random walks starting at the pink vertex, yellow vertex, and green vertex, respectively. These four random walks (starting at four different vertices) are not completely independent. For any pair of random walks, before they meet, they move on the graph independently of each other; whenever they meet, they move together henceforth. Concretely, backwards-chains 2 and 3 meet at $(\text{yellow}, 7)$ and these chains are identical starting from block $(\text{yellow}, 7)$; this holds similarly for other pairs of backwards chains. Finally, these four backward chains all meet at the block $(\textbf{green}, 4)$ and move together henceforth. Notably, this block is exactly the last block in the maximal common prefix of the four longest chains of round 8. Thus, the maximal inconsistency among the longest chains of round 8 is identical to the number of backwards steps it takes for all these four random walks to coalesce into one. This relation is not a coincidence. It can be shown (detailed in the proof of Theorem 8) that this identity holds for general n. Formal proof of Theorem 8 can be found in Appendix 7.

4.2 General p: Adversary-Free

The analysis for general p is significantly more challenging than that of $p = 1$ in two ways: (1) we need to repeatedly apply coupling arguments; and (2) we need to characterize the coalescence time of a new notion of coalescing random walks (the lazy coalescing random walks), the latter of which could be of independent interest for a broader audience.

▶ **Theorem 12.** *Suppose that $np = \Omega(1)$. If $p < \frac{4\ln 2}{n}$, in expectation, at the end of round t, the local chains at the nodes share a common prefix of length $(1 + (1-(1-p)^n)\,t) - O(\frac{1}{npe^{-np}})$. If $p \geq \frac{4\ln 2}{n}$, in expectation, at the end of round t, the local chains at the nodes share a common prefix of length $(1 + (1-(1-p)^n)\,t) - O\left(\frac{2np}{(1-2\exp(-\frac{1}{3}np))}\right)$.*

▶ Remark 13. The expression of the common prefix length in Theorem 12 contains two terms with the first term (i.e., $(1 + (1-(1-p)^n)\,t)$) being the only term that involves t. Intuitively, from this term, we can read out the common prefix length growth rate w.r.t. t. The second term (which is expression in terms of Big-O notation) can be interpreted as a quantification of the maximal inconsistency of the honest chains.

[5] Formally shown in the proof of Theorem 8 via introducing an auxiliary process.

Now we further interpret these two terms via simplifying the expression using the inequalities $(1 - np) \leq (1 - p)^n \leq \exp(-np)$.

(1) When $np = o(1)$, it is true that $(1 - p)^n \approx (1 - np)$ for large n, which implies that $(1 - (1 - p)^n)\, t \approx npt = o(t)$, i.e., the common prefix grows at a speed $o(t)$. The maximal inconsistency bound $O(\frac{1}{npe^{-np}})$ is not tight. Nevertheless, via a straightforward calculation, we know that the maximal inconsistency is $O(1)$.

(3) When $np = \omega(1)$, we have $0 \leq (1 - p)^n \leq \exp(-np) \to 0$ as $np \to \infty$. Thus the common-prefix grows at the speed $(1 - (1 - p)^n)\, t \approx t = \Omega(t)$ with maximal inconsistency $O(np)$ for sufficiently large np.

(4) When $np = c \in (0, 1)$, it is true that $(1 - p)^n = (1 - c/n)^n \to \exp(-c)$ as $n \to \infty$. The common-prefix grows at the speed of $\Theta(t)$ for sufficiently large n and the maximal inconsistency is $O(1)$.

Overall, when np gets larger, the common-prefix growth increases and the maximal inconsistency grows at a much slower rate.

The following definition and lemma are used in proving Theorem 12. This lemma could be of independent interest to a broader audience and its proof can be found in the appendix.

▶ **Definition 14** (Lazy coalescing random walk)**.** *For any fixed $u \in (0, 1)$, we say n* particles *are u-lazy coalescing random walks if for each step: with probability $(1 - u)$, each particle stays at its current location; with probability u, each particle moves to an adjacent vertex picked uniformly at random. If two or more particles meet at a location, they unite into a single particle and continue the procedure. The* coalescence time *is the same as that in Definition 10.*

▶ **Lemma 15.** *Suppose that G is a complete graph of size $|V| = n_g$ (where $n_g \geq 2$) with self-loops. For any $u \in (0, 1)$, the coalescence time of the u-lazy coalescing random walks is $C_G(n_g) = O(n_g/u)$.*

Proof Sketch of Theorem 12. When $p < \frac{4 \ln 2}{n}$, we can use Poisson approximation to approximate the distribution of number of blocks in each round. A straightforward calculation shows that the probability of having exactly one block in a round is $np \exp(-np)$. Thus, in expectation, the maximal inconsistency is $O\left(\frac{1}{np \exp(-np)}\right)$. Henceforth, we restrict our attention to the setting where $p \geq \frac{4 \ln 2}{n}$ and quantify the expected maximal inconsistency among the longest chains of round t. It is attempting to apply arguments similar to that in the proof of Theorem 8 and derive a bound on the maximal inconsistency via stochastic dominance. However, the obtained bound on the maximal inconsistency is $O(n)$ which could be extremely loose for a wide range of p. Nevertheless, based on the insights obtained in this coarse analysis, we can come up with a much finer-grained analysis and obtain the bound in Theorem 12. Similar to the proof of the special case when $p = 1$, in our fine-grained analysis for general $p \in (0, 1)$, we couple the growth of the common prefix in Nakamoto protocols with the coalescing time random walks on complete graphs. The major differences from the proof of $p = 1$ are: (1) instead of the standard coalescing random walks, we need to work with a lazy version of it, formally defined in Definition 14; (2) there is no fixed correspondence between a color and a node – in our proof of general p, the correspondence is round-specific rather than fixed throughout the entire dynamics; (3) there is no bijection between a sample path of the Nakamoto dynamics and that of the backwards coalescing random walks, thus, we need to rely on stochastic dominance to build up the connection of these two dynamics.

4.3 General p: Adversary-Prone

Throughout this section, we assume $p < 1$. In this subsection, we consider adversary-prone systems, i.e., $b > 0$. Simple concentration arguments show that when $bp \geq (1 + 2c)$ for any given $c \in (0, 1)$, using vanilla Nakamoto consensus the chain quality could be near zero. To make larger p feasible, we introduce a new assumption – Assumption 16 – which we then remove in Section 5 by providing a construction that ensures Assumption 16 with all but negligible probability. Specifically, we use a cryptographic tool called a VDF to ensure that over a sufficiently long time window, the corrupt nodes can only collectively extend a chain by more than one block in a round with negligible probability.

▶ **Assumption 16.** *In each round, a chain can be extended by at most 1 block.*

To strengthen the protocol robustness, we make the additional minor modification requiring each honest node to selectively relay chains at the *beginning* of a round.

Selective relay rule. At each honest node i, for each iteration $t \geq 1$: Node i looks at the chains it received in the previous round $t - 1$, and if any of them are longer than its own local longest chain, it not only chooses one of the longest chains to replace its local one, it also broadcasts it to other nodes before it begins mining in round t.

As implied by our proof, this modification can reduce the maximal difference between the lengths of the longest chains kept by the honest nodes and by the corrupt nodes. Intuitively, if the adversary sends two chains of different lengths to two different groups of honest nodes, with the selective relay rule, only the longer chain would survive in this round. Notably, it is possible that none of them survive in this round. Even with the assurance guaranteed by Assumption 16, compared with the adversary-free settings, the analysis for the adversary-prone setting is challenging. This is because the corrupt nodes could deviate from the specified symmetry breaking rule. For example, a corrupt node can choose not to extend its longest chain, or can choose from its set of longest chains in any way that provides advantage. In addition, a corrupt node can hide blocks it has mined from the honest nodes for as long as it wants, or from some subset of the honest nodes during a round.

For simplicity and for technical convenience, we assume that a corrupt node randomly chooses among longest chains that end with an honest block. This assumption is only imposed in the rare event when simultaneously both the adversary has no adversary advantage (see Definition 17) and only honest nodes mine blocks in the most recent nonempty round.

In contrast to the adversary-free setting where the lengths of honest nodes' local chains differ by at most 1, in the presence of an adversary, such difference could be large. To precisely bound this difference, we introduce a random process we call *adversary advantage*:

▶ **Definition 17** (Adversary advantage). *Let $\{\mathcal{N}(t)\}_{t=0}^{\infty}$ be the random process defined as*

- $\mathcal{N}(0) = 0$*, and*
- *for $t \geq 1$,*

$$\mathcal{N}(t) = \begin{cases} \mathcal{N}(t-1) + 1, & \textit{if only corrupt nodes found blocks in round } t; \\ \max\{\mathcal{N}(t-1) - 1,\ 0\}, & \textit{if only honest nodes found blocks in round } t; \\ \mathcal{N}(t-1), & \textit{otherwise.} \end{cases}$$

Note that the random process $\{\mathcal{N}(t)\}_{t=0}^{\infty}$ is independent of the adversarial behaviors of the corrupt nodes. To make the discussion concrete, we introduce the following definition.

▶ **Definition 18.** *The length of the longest chains kept by the honest nodes* **at round** t *is defined as the length of the longest local chains kept by honest nodes* at the end *of round* t*.*

▶ **Lemma 19.** *For any* $t \geq 1$*, at the end of round* t*, the length of the longest chains kept by the adversary – henceforth referred to as an adversarial longest chain of round* t *– is at most* $\mathcal{N}(t)$ *longer than the length of a chain kept by an honest node.*

Proof of Lemma 19 can be found in the full version. From its proof, we can deduce an attacking strategy of the adversary that meets the upper bound in Lemma 19. The following lazy random walk, referred to as *coalescing opportunities*, is important in our analysis. It can also be used to quantify the chain quality.

▶ **Definition 20.** *Let* $t_1, t_2, \cdots$ *be the rounds in which at least one node mines a block with the understanding that* $t_0 = 0$*. Let* $\mathcal{J}(m)$ *be a random walk defined as*

$$\mathcal{J}(m) = \begin{cases} 0, & \text{if } m = 0; \\ \mathcal{J}(m-1)+1, & \text{if only honest nodes mine a block during round } t_k; \\ \mathcal{J}(m-1)-1, & \text{if only corrupt nodes mine a block during round } t_k; \\ \mathcal{J}(m-1), & \text{otherwise.} \end{cases}$$

▶ Remark 21. A couple of interesting facts on the coalescing opportunities dynamics are: Among the most recent m blocks in a longest chain, there are at least $\mathcal{J}(m)$ blocks mined by the honest nodes. In addition, regardless of the behaviors of the adversary, for any two longest chains, there are at least $\mathcal{J}(m)$ block positions each of which has non-zero probability of being in the common prefix of these two chains.

Let $p_{+1} = \mathbb{P}\{\mathcal{J}(m) = \mathcal{J}(m-1)+1\}$ and $p_{-1} = \mathbb{P}\{\mathcal{J}(m) = \mathcal{J}(m-1)-1\}$, i.e., p_{+1} (resp. p_{-1}) is the probability for $\mathcal{J}(m)$ to move up (resp. down) by 1. We have

$$p_{+1} = \frac{(1-p)^b\left(1-(1-p)^{n-b}\right)}{1-(1-p)^n} \quad \text{and} \quad p_{-1} = \frac{\left(1-(1-p)^b\right)(1-p)^{n-b}}{1-(1-p)^n}. \tag{2}$$

It is easy to see that when $b > \frac{1}{2}n$, it holds that $p_{+1} > p_{-1}$. For ease of exposition, let $p^* = \mathbb{P}\{\mathcal{J}(t) \neq \mathcal{J}(t-1)\} = p_{+1} + p_{-1}$.

▶ **Lemma 22.** *With probability at least* $\left(1 - \exp\left(-\frac{(p_{+1}-p_{-1})^2 M}{16p^*}\right) - \exp\left(-\frac{(p^*)^2 M}{2}\right)\right)$*, it holds that* $\mathcal{J}(M) \geq \frac{(p_{+1}-p_{-1})M}{4}$*.*

Lemma 22 gives a high probability lower bound on the number of coalescing opportunities during M nonempty rounds.

▶ **Theorem 23.** *For any given* $T \geq 1$ *and* $M \geq \frac{4}{\beta(p_{+1}-p_{-1})}$ *where* $\beta = \frac{(n-b)p}{2(3np)^2}$*, at the end of round* T*, with probability at least*

$$1 - \exp\left(-\frac{(p^*)^2 M}{2}\right) - \exp\left(-\frac{(p_{+1}-p_{-1})^2 M}{16p^*}\right) - \frac{2}{\beta}\exp\left(-\frac{1}{2}(n-b)\right)$$

over the randomness in the block mining, the expected maximal inconsistency among a given pair of honest nodes is less than M*, where the expectation is taken over the randomness in the symmetry breaking.*

▶ Remark 24. It is worth noting that $\beta = \frac{(n-b)p}{2(3np)^2} = \frac{1}{18}\frac{(n-b)}{n}\frac{1}{np}$, i.e., β is a function of the fraction of honest nodes and the total mining power of the nodes in the system.

Suppose that $n \geq 2\log\frac{4}{\epsilon\beta}$ for any given $\epsilon \in (0,1)$. Let

$$M^* = \max\left\{\frac{4\log 1/\epsilon}{(p^*)^2}, \frac{4}{\beta(p_{+1}-p_{-1})}, \frac{16p^*}{(p_{+1}-p_{-1})^2}\log\frac{4}{\epsilon}\right\}.$$

From Theorem 23, we know that with probability at least $1-\epsilon$, the maximal inconsistency is less than M^*. Roughly speaking, when b gets smaller, M^* mainly gets smaller.

Proof of Theorem 23. We use N_t to denote the number of blocks generated during round t and associate each node with a distinct color in $\{c_1, \cdots, c_n\}$. If node i mines a block during round t, we use (c_i, t) to denote this block. The genesis block is denoted as $(c_1, 0)$. Recall that the blocks mined during round t are collectively referred to as the block layer t. As the randomness in the block generation (i.e., puzzle solving of individual nodes) is independent of the adversarial behaviors of the corrupt nodes and is independent of which chain an honest node chooses to extend, we consider the auxiliary process wherein the nodes mine blocks for the first T rounds, and then the corrupt nodes and honest nodes sequentially decide on block attachments. Let $\{i_1, \cdots, i_K\}$ be the set of rounds such that $N_{i_k} \neq 0$ for each $i_k \in \{i_1, \cdots, i_K\}$. Let j_1 and j_2 be any two honest nodes whose chains at the end of round T are denoted by $C_1(T)$ and $C_2(T)$, respectively. For each of these chains, we can read off a sequence of colors

$$\text{for Chain } C_1(T):\ c_1c(1,2)c(1,3)\cdots c(1,\ell_1), \quad \text{and}$$
$$\text{for Chain } C_2(T):\ c_1c(2,2)c(2,3)\cdots c(2,\ell_2),$$

where ℓ_1 and ℓ_2, respectively, are the lengths of chains $C_1(T)$ and $C_2(T)$, c_1 is the color of the genesis block, $c(1,k)$ for $k \in \{2, \cdots, \ell_1\}$ is the color of the k–th block in $C_1(T)$ and $c(2,k)$ for $k \in \{2, \cdots, \ell_2\}$ is the color of the k–th block in $C_2(T)$. If $\ell_1 \neq \ell_2$, without loss of generality, we consider the case that $\ell_1 < \ell_2$; the other case can be handled similarly. We augment the color sequence $c_1c(1,2)c(1,3)\cdots c(1,\ell_1)$ to the length ℓ_2 sequence as

$$c_1c(1,2)c(1,3)\cdots c(1,\ell_1)c(1,\ell_1+1)\cdots c(1,\ell_2),$$

by setting $c(1,k) = c_0$ for $k = \ell_1+1, \cdots, \ell_2$ where $c_0 \notin \{c_1, \cdots, c_n\}$ is a special color that never shows up in a real block. It is easy to see that $C_1(T)$ and $C_2(T)$ *start* to be inconsistent at their k-th block if and only if $c(1,k') \neq c(2,k')$ for each $k' \in \{k, \cdots, \ell_2\}$. Let $\{i_{h_1}, \cdots, i_{h_R}\} \subseteq \{i_1, \cdots, i_K\}$ such that for each $i_{h_r} \in \{i_{h_1}, \cdots, i_{h_R}\}$ it holds that

- Only honest nodes successfully mined blocks;
- $\mathcal{N}(i_{h_r-1}) = 0$.

For ease of exposition, we refer to each of i_{h_r} as *a coalescing opportunity.* Recall that each of the honest nodes extends one of the longest chains it receives. By Lemma 19, we know that each of $C_1(T)$ and $C_2(T)$ contains a block generated during round i_{h_r}. Let (c_1', i_{h_r}) and (c_2', i_{h_r}) be the blocks included in $C_1(T)$ and $C_2(T)$, respectively. If (c_1', i_{h_r}) is in the k-th position in $C_1(T)$, then (c_2', i_{h_r}) is also in the k-th position in $C_2(T)$. For each i_{h_r}, we denote the set of chains (including the forwarded chains) received by j_1 and j_2 at round i_{h_r}, denoted by $\mathcal{C}_1^r$ and $\mathcal{C}_2^r$. Since the adversary can hide chains to a selective group of honest nodes, $\mathcal{C}_1^r$ and $\mathcal{C}_2^r$ could be different. The probability of j_1 and j_2 extending the same chain at round i_{h_r} is

$$\frac{|\mathcal{C}_1^r \cap \mathcal{C}_2^r|}{|\mathcal{C}_1^r|\,|\mathcal{C}_2^r|} \geq \frac{\mathrm{NB}(i_{h_r-1})}{\left(\mathrm{NB}(i_{h_r-1}) + \mathrm{AB}(i_{h_r-1}) + \mathrm{AB}(\widetilde{i_{h_r-1}})\right)^2} \tag{3}$$

where the inequality follows from Lemma 33 of the the full version. By Lemma 22, we know that in the M non-empty block layers that are most recent to round T,

$$R \geq \mathcal{J}(M) \geq \frac{(p_{+1} - p_{-1})M}{4}$$

holds with probability at least $\left(1 - \exp\left(-\frac{(p^*)^2 M}{2}\right) - \exp\left(-\frac{(p_{+1}-p_{-1})^2 M}{16p^*}\right)\right)$. In addition, it can be shown that for each of the r ensured by Lemma 22 we have

$$\max\{|\mathcal{C}_1^r|, |\mathcal{C}_2^r|\} \leq \mathrm{NB}(i_{h_r-1}) + \mathrm{AB}(i_{h_r-1}) + \mathrm{AB}(\widetilde{i_{h_r-1}})\mathbb{1}\{\mathrm{AB}(i_{h_r-1}) = 0\}.$$

For any i_k, let X_k be the number of blocks mined by the honest nodes during round i_k such that $X_k \neq 0$. Using conditioning and Hoeffding's inequality, the following holds with probability at least $\left(1 - 2\exp\left(-\frac{1}{2}(n-b)\right)\right)$,

$$X_k \geq \frac{1}{2}(n-b)p \quad \text{and} \quad X_k + Y_k + Y_{k-1}\mathbb{1}\{Y_k = 0\} \leq 3np,$$

which implies that $\frac{X_k}{X_k + Y_k + Y_{k-1}\mathbb{1}\{Y_k=0\}} \geq \frac{(n-b)p}{2(3np)^2} \triangleq \beta$. On average over the random symmetry breaking, it takes at most $1/\beta$ coalescing opportunities backwards for chains $C_1(T)$ and $C_2(T)$ to coalesce into one. Thus, we need $\frac{(p_{+1}-p_{-1})M}{4} \geq \frac{1}{\beta}$. ◀

5 VDF-Based Scheme

In this section, we present a scheme to ensure Assumption 16. The key cryptographic tool we use in the following scheme is the construction of the *verifiable delay function*, $\mathcal{F}(x)$, which we define informally below. Please refer to [4] for the formal definition (also defined formally in the full version of our paper).

▶ **Definition 25** (Verifiable Delay Function (informal)). *Let λ be our security parameter. There exists a function $\mathcal{F}$ with difficulty $X = O(poly(\lambda))$ where the output $y \leftarrow \mathcal{F}(x)$ (where $x \in \{0,1\}^\lambda$) cannot be computed in less than X sequential computation steps, even provided $poly(\lambda)$ parallel processors, with probability at least $1 - \mathsf{negl}(\lambda)$. The VDF output can be verified, quickly, in $O(\log(X))$ time.*

We set the difficulty of the VDF to the duration of a round; in other words, the difficulty is set such that the VDF produces exactly one output at the end of each round. We amend default Nakamoto consensus by adding the following procedure. We believe this could be added in a backwards-compatible way to existing Nakamoto implementations, like Bitcoin. Backwards-compatibility is desirable in decentralized networks because it means that a majority of the network can upgrade to the new protocol and non-upgraded nodes can still verify blocks and execute transactions. Below we describe a scheme that, when added to Nakomoto consensus, assures Assumption 16. The proof of the following theorem is in the full version of our paper.

▶ **Theorem 26.** *Assumption 16 is satisfied by our VDF-based scheme.*

VDF-Scheme Overview. The VDF-scheme works intuitively as follows. We number the rounds beginning with round 0. All nodes have the genesis block B_0 in their local chains in round 0 and starting mining blocks in round 1. In round 0, the VDF output is computed using 0 as the input. During each round $j > 0$, each node computes a VDF output, y_j, (using $\mathcal{F}$) for the current round j where the input to $\mathcal{F}$ is the output of the VDF, y_{j-1}, from the previous round concatenated with the round number, j. Both inputs are necessary; the output of the VDF from the previous round ensures that we cannot compute the VDF output for this round until we have obtained the output for the previous round, and the round number is necessary to ensure that the output is *not* used for a future round. Once the VDF output is computed, each honest node attempts to mine a block using the VDF output as part of the input to the mining attempt. This also ensures that the block generation rate of honest nodes is upper bounded by np. Then, each node which successfully mines a block sends the new chain to all other nodes.

All honest nodes verify that each chain satisfies two conditions:

1. Let $o_1, \ldots, o_\ell$ be the VDF outputs contained in blocks $B_1, \ldots, B_\ell$, respectively, of a chain C (the genesis block does not contain a VDF output). Let $r_1, \ldots, r_\ell$ be the rounds where $o_1, \ldots, o_\ell$ were computed, respectively. Then, $r_1 < \cdots < r_{\ell-1} < r_\ell$.
2. o_i is the VDF output computed from round $r_i \geq i - 1$.

The honest nodes also check all proofs included in the chains, confirming that the VDF outputs are correctly computed and the blocks are correctly mined using the VDF outputs. An honest node discards any chain which does not pass verification.

Pseudocode. The precise pseudocode of our VDF-based scheme is given below. Using $\mathcal{F}$, each honest node i performs the following:

1. Initially, all honest nodes use input 0 at the start of the protocol to obtain output $y_0 = \mathcal{F}(0)$ for round 0.
2. Let $d_j = \mathcal{F}(y_{j-1})$ be the output of the VDF for round j and $y_j = d_{j-1}|j$.[6] i stores y_j.
3. When i mines a block B_j, i includes the output $y_{j-1} = d_{j-1}|j$ from the previous round in B_j, ie. B_j is mined with y_{j-1} as part of the input.
4. Each node which successfully mines a block adds the mined block to its local chain. Then, it broadcasts its local chain to all other nodes.
5. For each longest chain received, each node verifies the following:
   a. Let $o_1, \ldots, o_\ell$ be the VDF outputs stored in each block in order starting with the first block and ending with the ℓ-th block. Let $r_1, \ldots, r_\ell$ be the rounds associated with the VDF output. Then, $r_\ell > r_{\ell-1} > \cdots > r_1$.
   b. The k-th block in the chain (starting from the genesis block) is mined using $y_{k'}$ from round $k' \geq k - 1$.
   c. The proofs of the VDF output and the mining output are correct, i.e. the block is correctly mined using the corresponding VDF output.
6. If i receives a chain where more than one block in the chain is mined with the same y_j (for any j smaller than the current round), the node discards the chain.
7. At the end of round j, i sets $y_{j+1} \leftarrow \mathcal{F}(y_j)|j+1$ and begins computing the next value $\mathcal{F}(y_{j+1})$ using y_{j+1} as input.

Due to space constraints, we do not include the proof of Theorem 26; please find the full proofs in the full version of our paper. However, the intuition for our proof is straightforward. Items 5a and 5b ensure that no chain accepted by an honest node contains more than one block per VDF output. Setting the difficulty of the VDF to the duration of the round ensures that at most one VDF output is produced during a round. Together, these two observations prove Theorem 26, namely, that any chain held by an honest node can be extended by at most one block each round.

[6] Here, $a|b$ is the commonly used notation indicating concatenation between a and b.

6 Discussion

Validation and Communication Costs. A higher p means a faster block rate and thus more blocks. The validation and bandwidth complexity of Nakamoto protocols are proportional to block size and the number of blocks that are mined, since each miner validates and then communicates every mined block to all other miners (in practice, nodes do not necessarily gossip shorter chains, and taking advantage of nodes' memory overlap can help reduce block transfer size [8]). One needs to determine the optimal value of p that trades off validation and bandwidth complexity and chain growth. This work expands the space of p to consider.

Other Symmetry-Breaking Strategies. Here we consider three other symmetry-breaking strategies with high p. *First-seen* is where all honest nodes take the first chain out of the longest-length chains they see, and *lexicographically-first* is where honest nodes take the lexicographically-first chain of the set of longest chains according to some predetermined ordering, for example alphabetically. Intuitively, the adversary can control the network and thus cause different honest nodes to see different chains of the same length first for first-seen, impacting common-prefix, or grind on blocks to always produce the lowest lexicographically-ordered chain for lexicographically-first, impacting chain-quality. A third strategy is to use a *global-random-coin*: Suppose that all nodes have access to a permutation oracle $\mathcal{P}$ that returns a permutation sampled uniformly at random of a number of elements passed into it *where any subset of elements obey the same partial ordering.* With $\mathcal{P}$ symmetry-breaking is trivial since all honest nodes will agree on the result of the coin flip. Furthermore, if the coin is fair, then the number of honest blocks added to the chain is proportional to the fraction of honest nodes. However, in reality, it is difficult and oftentimes infeasible to ensure such a strong guarantee.

Conclusion. In this work we show that unlike previously thought, convergence opportunities are not necessary to make chain progress. We use *coalescing random walks* to analyze the correctness of Nakamoto consensus under a regime of puzzle difficulty previously thought to be untenable, expanding the space of p for protocol designers.

References

1 David Aldous and Jim Fill. Reversible markov chains and random walks on graphs, 2002.

2 Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 585–602, 2019.

3 Erica Blum, Aggelos Kiayias, Cristopher Moore, Saad Quader, and Alexander Russell. *The Combinatorics of the Longest-Chain Rule: Linear Consistency for Proof-of-Stake Blockchains*, pages 1135–1154. SIAM, 2020. doi:10.1137/1.9781611975994.69.

4 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, pages 757–788, 2018. doi:10.1007/978-3-319-96884-1_25.

5 Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. Cryptology ePrint Archive, Report 2018/601, 2018. URL: https://eprint.iacr.org/2018/601.

6 Colin Cooper, Robert Elsasser, Hirotaka Ono, and Tomasz Radzik. Coalescing random walks and voting on connected graphs. *SIAM Journal on Discrete Mathematics*, 27(4):1748–1758, 2013.

7 Colin Cooper, Alan Frieze, and Tomasz Radzik. Multiple random walks in random regular graphs. *SIAM Journal on Discrete Mathematics*, 23(4):1738–1761, 2010.

8 Matt Corallo. Compact block relay, 2016. URL: `https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki`.

9 Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual international cryptology conference*, pages 139–147. Springer, 1992.

10 EOS. v2.0 consensus protocol, 2021. URL: `https://developers.eos.io/welcome/v2.0/protocol/consensus_protocol`.

11 Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

12 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015*, pages 281–310, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

13 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

14 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. Full analysis of nakamoto consensus in bounded-delay networks. Cryptology ePrint Archive, Report 2020/277, 2020. URL: `https://eprint.iacr.org/2020/277`.

15 Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323. Springer, 2017. `doi:10.1007/978-3-319-63688-7_10`.

16 Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. *IACR Cryptol. ePrint Arch.*, 2015:1019, 2015. URL: `http://eprint.iacr.org/2015/1019`.

17 Lucianna Kiffer, Rajmohan Rajaraman, and abhi shelat. A better method to analyze blockchain consistency. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 729–744, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3243734.3243814`.

18 Silvio Micali. Algorand 2021 performance, 2020. URL: `https://www.algorand.com/resources/blog/algorand-2021-performance`.

19 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

20 Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system.(2008), 2008.

21 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673, Cham, 2017. Springer International Publishing.

22 Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 643–673. Springer, 2017.

23 Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 315–324, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3087801.3087809`.

24 Ling Ren. Analysis of nakamoto consensus. *IACR Cryptol. ePrint Arch.*, 2019:943, 2019. URL: `https://eprint.iacr.org/2019/943`.

25 Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer, 2016.

26 R. Zhang and B. Preneel. Lay down the common metrics: Evaluating proof-of-work consensus protocols' security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 175–192, 2019. doi:10.1109/SP.2019.00086.

27 Jun Zhao, Jing Tang, Zengxiang Li, Huaxiong Wang, Kwok-Yan Lam, and Kaiping Xue. An analysis of blockchain consistency in asynchronous networks: Deriving a neat bound. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 179–189. IEEE, 2020. doi:10.1109/ICDCS47774.2020.00039.

7 Proof of Theorem 8

Proof of Theorem 8. We formalize the arguments of the main proof ideas in Section 4.1. Let $\{c_1, \cdots, c_n\}$ be a set of n different colors. We associate each node in the system with a color. We use (c_i, t) to denote the block generated by honest node i during round t and $(c_0, 0)$ to denote the genesis block. We use $(c_i, t) \to (c_{i'}, t-1)$ to denote the event that block (c_i, t) is attached to block $(c_{i'}, t-1)$, which occurs with probability $\frac{1}{n}$ under our symmetry-breaking rule. To quantify the maximal inconsistency of the longest chains of round T, we consider the following auxiliary random process. It can be easily shown that there is a bijection between the sample paths of the Bitcoin blockchain protocol and the sample paths of this auxiliary process, and that the auxiliary process and the original blockchain protocol with random symmetry breaking have the same probability distribution.

Auxiliary random procedure: For any given $T \geq 1$, do the following:

(i) Let each color generate a block for each of the rounds in $\{1, 2, \cdots, T\}$;

(ii) Attach each of the block $(c_i, 1)$ for $i = 1, \cdots, n$ to the genesis block $(c_0, 0)$;

(iii) For each $t \geq 2$ and each (c_i, t), attach it to one of the blocks $\{(c_i, t-1), i = 1, \cdots, n\}$ uniformly at random (i.e., with probability $1/n$).

Connecting to coalescing random walks: Here, we formally quantify the connection between the maximal inconsistency among the longest chains of round T with the coalescing time of n random walks on an n-complete graph. Since $p = 1$ and there is no adversary, the number of longest chains received by each honest node at each round is n. Let $C(T, c_1), \cdots, C(T, c_n)$ be the n longest chains of round T ending with blocks $(c_1, T), \cdots, (c_n, T)$, respectively. We first show that each of these n chains can be coupled with a random walk on the n-complete graph. Without loss of generality, let's consider $C(T, c_1)$ which can be expanded as

$$C(T,c_1) := (c_0,0) \leftarrow (c_{i_1},1) \leftarrow \cdots \leftarrow \left(c_{i_{t-1}}, t-1\right) \leftarrow (c_{i_t}, t) \leftarrow \cdots \leftarrow \left(c_{i_{T-1}}, T-1\right) \leftarrow (c_1, T), \qquad (4)$$

where c_t is the color of the $(t+1)$-th block in the chain. Note that the chain $C(T, c_1)$ is random because the sequence of block colors $c_0 c_{i_1} \cdots c_{i_{t-1}} c_{i_t} \cdots c_{i_{T-1}} c_1$ is random. Moreover, the randomness in $C(T, c_1)$ is fully captured in the randomness of the block colors. We have

$$\begin{aligned}
&\mathbb{P}\left\{C(T,c_1) = (c_0,0) \leftarrow (c_{i_1},1) \leftarrow \cdots \leftarrow \left(c_{i_{t-1}}, t-1\right) \leftarrow (c_{i_t}, t) \leftarrow \cdots \leftarrow \left(c_{i_{T-1}}, T-1\right) \leftarrow (c_1, T)\right\} \\
&\overset{(a)}{=} \mathbb{P}\left\{(c_0, 0) \leftarrow (c_{i_1}, 1)\right\} \prod_{t=2}^{T} \mathbb{P}\left\{\left(c_{i_{t-1}}, t-1\right) \leftarrow (c_{i_t}, t)\right\} \\
&= \prod_{t=2}^{T} \mathbb{P}\left\{\left(c_{i_{t-1}}, t-1\right) \leftarrow (c_{i_t}, t)\right\},
\end{aligned}$$

where the last equality is true as $\mathbb{P}\{(c_0, 0) \leftarrow (c_{i_1}, 1)\} = 1$, and the equality (a) holds because under our symmetry-breaking rule, neither the previous history up to round t nor the future block attachment choices after round t affects the choice of the chain extension in round t.

Moreover, the probability of any realization of the color sequence $c_0 c_{i_1} \cdots c_{i_{t-1}} c_{i_t} \cdots c_{i_{T-1}} c_1$ (i.e., a sample path on the block colors in Bitcoin) is $\left(\frac{1}{n}\right)^{T-1}$. Let's consider the complete graph with vertex set $\{c_1, c_2, \cdots, c_n\}$. Under our symmetry breaking rule, the backwards color sequence $c_1 c_{i_{T-1}} \cdots c_{i_t} c_{i_{t-1}} \cdots c_{i_1}$ (without considering the genesis block) is a random walk on the n-complete graph starting at vertex c_1. Similarly, we can argue that $C(T, c_2), \cdots, C(T, c_n)$ correspond to $n-1$ random walks on the n-complete graphs starting at vertices $c_2, \cdots, c_n$, respectively. As argued in the **main proof ideas** paragraph, these n random walks are not fully independent. In fact, they are coalescing random walks, and their coalescence is exactly the maximal inconsistency among the longest chains of round T.

With the above connection of the longest chain protocol augmented by uniformly-at-random symmetry breaking with coalescing random walks. We conclude by applying Theorem 11. ◀

8 Proof of Lemma 15

Proof of Lemma 15. To characterize the coalescence time, similar to the analysis in [6], for any given $k \in \{1, \cdots, n_g\}$, we construct a larger graph $Q = Q_k = (V_Q, E_Q)$, where $V_Q = V^k$ and two vertices $\boldsymbol{v}, \boldsymbol{w} \in V^k$ if $\{v_1, w_1\}, \cdots, \{v_k, w_k\}$ are edges of G. Let M_k be the time until the first meeting in the original graph G. Let $S \subseteq V_Q$ denote the set of all possible configurations of the locations of the n_g random walks at the first meeting,

$$S_k = \{(v_1, \cdots, v_k) : v_i = v_j \quad \text{for some } 1 \leq i < j \leq k\}. \tag{5}$$

It is easy to see that there is a direct equivalence between the u-lazy random walks on G and the single u-lazy random walk on Q. Since Q is a complete graph with self-loops, the limiting distribution of lazy random walk on Q is the same as the standard random walk on Q. Let $\boldsymbol{\pi}^Q \in \mathbb{R}^{|V^k|}$ be the stationary distribution of a standard random walk on Q and let $\pi^Q_{S_k} = \sum_{\boldsymbol{v} \in S_k} \pi^Q_{\boldsymbol{v}}$. By [6, Lemma 4], we know that for any $1 \leq k \leq k^*$ where $k^* \triangleq \max\{2, \log n_g\}$, it holds that

$$\pi^Q_{S_k} \geq \frac{k^2}{8n_g}.$$

Let $H_{\boldsymbol{v}, S_k}$ denote the hitting time of vertex set S_k starting from vertex v and let

$$H^Q_{\boldsymbol{\pi}}(H_{S_k}) = \sum_{\boldsymbol{v} \in V^k} \boldsymbol{\pi}^Q_{\boldsymbol{v}} H_{\boldsymbol{v}, S_k}$$

denote the expected hitting time of S_k from the stationary distribution $\boldsymbol{\pi}^Q$. From [1, Lemma 2.1] and the fact we can contract the vertex set S_k into one pseudo vertex, similar to [6, proof of Theorem 2], we have that

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\pi}^Q}[H_{S_k}] &= \frac{\sum_{t=0}^{\infty}\left(P^t_{S_k}(S_k) - \boldsymbol{\pi}^Q_{S_t}\right)}{\boldsymbol{\pi}^Q_{S_k}} = \frac{\sum_{t=0}^{\infty}\left((1-u)^t + (1-(1-u)^t)\,\boldsymbol{\pi}^Q_{S_k} - \boldsymbol{\pi}^Q_{S_k}\right)}{\boldsymbol{\pi}^Q_{S_k}} \\
&\leq \frac{8n_g}{k^2}\frac{1}{u}\left(1 - \boldsymbol{\pi}^Q_{S_k}\right) \leq \frac{8n_g}{uk^2}.
\end{aligned}$$

In addition, by conditioning on whether the particles stay at their initial locations or not, we have

$$\mathbb{E}[M_k] = (1-u)(1 + \mathbb{E}[M_k]) + u\left(1 + \mathbb{E}_{\boldsymbol{\pi}^Q}(H_{S_k})\right),$$

which implies that

$$\mathbb{E}\left[M_k\right] \leq \frac{1}{u}\left(1+\frac{8n_g}{k^2}\right) = O\left(\frac{n_g}{uk^2}\right).$$

Thus, for any k such that $1 \leq k \leq k^* = \{2, \log n_g\}$, we have

$$\mathbb{E}\left[C_k\right] \leq \sum_{s=2}^{k} \mathbb{E}\left[M_s\right] \leq O(n_g/u).$$

Let $\mathcal{W}_u$ be a lazy random walk on the complete graph G with initial location u. In each round, with probability $(1-u)$, $\mathcal{W}_u$ stays at its current location and with probability u it moves to one of the current neighbors (including self-loops) uniformly at random. Let $\boldsymbol{\pi}^G$ the limiting distribution of the location vertex of $\mathcal{W}_u$. By [6, Eq.(8)], its mixing time is $t_{mix} = \frac{3\log n_g}{\log(1/(1-u))}$, i.e., for any given $u \in V$, when $t \geq \lceil \frac{3\log n_g}{\log(1/(1-u))} \rceil$,

$$\begin{aligned}
\|P_u^t - \boldsymbol{\pi}^G\|_1 &= \sum_{v\in V} \left|P_u^t(v) - \boldsymbol{\pi}_v^G\right| \\
&= \left|1-\pi_u^G\right|(1-u)^t + \sum_{v: v\in V, v\neq u} \left|\left(1-(1-u)^t\right)\boldsymbol{\pi}_v^G - \boldsymbol{\pi}_v^G\right| \\
&\leq 2(1-u)^t \leq \frac{2}{n_g^3} \leq \frac{1}{n_g^2}.
\end{aligned}$$

Here, with a little abuse of notation, we use P_u^t to denote the distribution of the state of $\mathcal{W}_u$ at round t. Let $t^* = k^* \log n_g \left(k^* t_{mix} + 3\mathbb{E}_{\boldsymbol{\pi}^Q}\left(H_{S_{k^*}}\right)\right)$. Following the arguments in [6, Section 5], we have

$$\begin{aligned}
C(n_g) &\leq 4t^* + \mathbb{E}\left[C_{k^*}\right] \\
&\leq 4\log n_g \left(k^* t_{mix} + 3\mathbb{E}_{\boldsymbol{\pi}^Q}\left(H_{S_{k^*}}\right)\right) + O(n_g/u) \\
&\leq \frac{4\log^4 n_g}{\log\frac{1}{1-u}} + 12\log^2 n_g \frac{8n_g}{u\log^2 n_g} + O(n_g/u) \\
&\leq \frac{4\log^4 n_g}{u} + \frac{96n_g}{u} + O(n_g/u) \\
&= O(n_g/u),
\end{aligned}$$

where the last inequality follows from $\log 1/(1-u) \geq u$. ◀

Time-Optimal Loosely-Stabilizing Leader Election in Population Protocols

Yuichi Sudo
Hosei University, Tokyo, Japan

Ryota Eguchi
Nagoya Institute of Technology, Japan

Taisuke Izumi
Osaka University, Japan

Toshimitsu Masuzawa
Osaka University, Japan

Abstract

We consider the leader election problem in the population protocol model. In pragmatic settings of population protocols, self-stabilization is a highly desired feature owing to its fault resilience and the benefit of initialization freedom. However, the design of self-stabilizing leader election is possible only under a strong assumption (i.e., the knowledge of the *exact* size of a network) and rich computational resource (i.e., the number of states). Loose-stabilization is a promising relaxed concept of self-stabilization to address the aforementioned issue. Loose-stabilization guarantees that starting from any configuration, the network will reach a safe configuration where a single leader exists within a short time, and thereafter it will maintain the single leader for a long time, but not necessarily forever. The main contribution of this paper is giving a time-optimal loosely-stabilizing leader election protocol. The proposed protocol with design parameter $\tau \geq 1$ attains $O(\tau \log n)$ parallel convergence time and $\Omega(n^\tau)$ parallel holding time (i.e., the length of the period keeping the unique leader), both in expectation. This protocol is time-optimal in the sense of both the convergence and holding times in expectation because any loosely-stabilizing leader election protocol with the same length of the holding time is known to require $\Omega(\tau \log n)$ parallel time.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases population protocols, leader election, loose-stabilization, self-stabilization

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.40

Related Version *Previous Version*: https://arxiv.org/abs/2005.09944

Funding This work was supported by JSPS KAKENHI Grant Numbers 19H04085 and 20H04140.

Acknowledgements We truly thank anonymous reviewers for their constructive and helpful comments. We also thank Przemyslaw Uznanski and Eric Severson: The first author of this paper got one of the key ideas of this paper after the discussion with them at PODC 2019.

1 Introduction

We consider the *population protocol* (PP) model [5] in this paper. A network called the *population* consists of n automata called *agents*. Pairs of agents execute *interactions* (i.e., pairwise communication) by which they update their states. These interactions are opportunistic, that is, they are unknown and unpredictable (or only predictable with probability). Agents are strongly anonymous: they do not have identifiers and cannot distinguish neighbors with the same state. As with the majority of studies on population protocols [5, 6, 4, 2, 14, 15, 22, 19], we assume that exactly one pair of agents is selected to have an interaction uniformly at random from all $\binom{n}{2}$ pairs at each step. In the PP model, time complexity such as expected

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 40; pp. 40:1–40:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Self/Loosely-stabilizing leader election in the PP model (The convergence/holding time is shown in expected parallel time).

	Type	Knowledge N	Convergence time	Holding time	#states	Design parameter
[10]	SS-LE	$N = n$	$O(n^2)$	∞	n	-
[9]	SS-LE	$N = n$	$O(n)$	∞	$O(n)$	-
[9]	SS-LE	$N = n$	$O(\log n)$	∞	$n^{O(n^{\log n})}$	-
[9]	SS-LE	$N = n$	$O(n^{\frac{1}{H+1}})$	∞	$n^{O(n^H)}$	$H = O(1)$
[18]	LS-LE	$n \leq N = O(n)$	$O(n)$	$\Omega(e^n)$	$O(n)$	-
[16]	LS-LE	$n \leq N = O(n)$	$O(n)$	$\Omega(e^n)$	$O(n)$	-
[22]	LS-LE	$n \leq N = \mathrm{poly}(n)$	$O(\tau \log^3 n)$	$\Omega(n^\tau)$	$O(\tau^2 \log^5 n)$	$\tau \geq 1$
ours	LS-LE	$n \leq N = \mathrm{poly}(n)$	$O(\tau \log n)$	$\Omega(n^\tau)$	$O(\tau \log n)$	$\tau \geq 1$

convergence time is usually evaluated in *parallel time*, that is, the number of steps divided by n (i.e., the number of agents). This is a natural measure of time because in practice, interactions typically occur in parallel in the population. For the remainder of this section, we presume parallel time when we discuss time complexity.

In this paper, we focus on the problem of self-stabilizing leader election (SS-LE). This problem requires that (i) starting from any configuration, a population reaches a safe configuration in which exactly one leader exists; and thereafter, (ii) it keeps this leader forever. These requirements guarantee tolerance against finitely many transient faults. Since many protocols (self-stabilizing or non-self-stabilizing) in the literature assume a unique leader [5, 7, 6], SS-LE is key to improving fault-tolerance of the PP model itself. However, it is known that no protocol can solve SS-LE unless every agent in the population knows the *exact size* n of the population [7, 10] [1]. Under this strong assumption (i.e., all agents know exact n), several SS-LE protocols have been presented in the literature. Cai et al. [10] gave the first SS-LE protocol under this assumption, which elects the unique leader within $O(n^2)$ time starting from any configuration. Recently, Burman et al. [9] gave three SS-LE protocols, which improve the convergence time at the cost of space complexity, that is, the number of states per agent (Table 1). For example, one of their protocols converges in $O(\log n)$ time but uses a super-exponential number of states.

We can discard the assumption of exact knowledge of n by slightly relaxing the requirement of self-stabilization, that is, by taking an approach called *loose-stabilization*. Loose-stabilization guarantees that the population reaches a safe configuration within a relatively short time starting from any initial configuration; after that, the specification of the problem (such as having a unique leader in the leader election) must be sustained for a sufficiently long time, though not necessarily forever. Sudo et al. [18] gave a loosely-stabilizing leader election (LS-LE) protocol by assuming that every agent knows a common *upper bound* N of n. Their protocol is not self-stabilizing; however, it is practically equivalent to an SS-LE protocol because it maintains the unique leader for an exponentially long time after reaching a safe configuration. Further, it converges in a safe configuration within $O(N)$ time starting from any configuration.[2] Hence, the convergence time is $O(n)$ if we have a good upper bound

[1] Strictly speaking, they prove a slightly weaker impossibility. However, we can prove this impossibility based on almost the same technique: a simple partitioning argument. See [22] for details (page 618, footnote).

[2] The convergence time of this protocol was proven to be $O(N \log n)$ in [18]. Later, it was found to be $O(N)$ according to Lemma 1 in [3].

$N = O(n)$. In practice, the knowledge of N is a much weaker assumption than the exact knowledge of n; the protocol works correctly even if we consider a large overestimation of n, such as $N = 100n$. Recently, Sudo et al. [22] gave an LS-LE protocol with poly-logarithmic convergence time, which has a design parameter τ (≥ 1) controlling the convergence and holding times. Given an upper bound $N \geq n$ such that $N = O(n^c)$ for some constant c, their protocol reaches a safe configuration within $O(\tau \log^3 n)$ time, and thereafter, it keeps the single leader for $\Omega(n^\tau)$ time, both in expectation.

Izumi [16] provided a lower bound on the convergence time of an LS-LE protocol, given that it keeps the unique leader for an exponentially long time after reaching a safe configuration. Sudo et al. [22] generalized this lower bound as follows: if the expected holding time of an LS-LE protocol is β/n, its expected convergence time must be $\Omega(\log \beta)$. Therefore, we have a gap of $\log^2 n$ factor between this lower bound and the upper bound given by Sudo et al. [22] when we require an expected holding time of $\Omega(n^\tau)$: the former is $\Omega(\tau \log n)$ and the latter is $O(\tau \log^3 n)$.

1.1 Our Contribution

We close the above-mentioned gap in this paper. That is, we develop an LS-LE protocol whose expected convergence time is $O(\tau \log n)$ and expected holding time is $\Omega(n^\tau)$, where $\tau \geq 1$ is the design parameter of the protocol. Interestingly, this convergence time is optimal for any length of holding time. For $\tau \geq 1$, we have no asymptotic gap between this convergence time and the lower bound given by Sudo et al. [22]. Even if a holding time of $o(n)$ is sufficient, the expected convergence time of our protocol with $\tau = 1$ remains optimal. This is because every LS-LE protocol requires $\Omega(\log n)$ time to reach a safe configuration regardless of the length of its holding time. Consider an execution of any LS-LE protocol starting from a configuration where all agents are leaders. Then, $n - 1$ agents must have at least one interaction before electing the unique leader. However, a simple analysis on the famous *coupon collector's problem* yields that this requires $\Omega(\log n)$ time (i.e., $\Omega(n \log n)$ steps) in expectation. In addition to time-optimality, the proposed protocol has a small space complexity: The number of states per agent is $O(\tau \log n)$, which is much smaller than $O(\tau^2 \log^5 n)$ in [22].

The proposed protocol also shows how useful loose-stabilization is in the PP model. When we set $\tau = 100$, its expected convergence time is $O(\log n)$, and the expected holding time is $\Omega(n^{100})$, practically forever. This protocol needs only the knowledge of N such that $n \leq N = O(n^c)$ holds for some constant c. Under self-stabilization, if we require the same convergence time, the only known solution [9] uses an super-exponential number of states and requires a much stronger assumption, i.e., the knowledge of *exact* n.

1.2 Further Related Work

Leader election has been extensively studied in the PP model. When we design non-self-stabilizing protocols, we can assume that all agents are in a specific state at the initial configuration. Leader election is then achieved by employing a simple protocol [5]. In this protocol, all agents are initially leaders, and we have only one transition rule: when two leaders meet, one of them becomes a follower (i.e., a non-leader). This simple protocol elects a unique leader in linear time and uses only two states at each agent. This protocol is time-optimal: Doty and Soloveichik [13] showed that any constant space protocol requires linear time to elect a unique leader. In a breakthrough result, Alistarh and Gelashvili [4] designed a (non-self-stabilizing) leader election protocol that converges in $O(\log^3 n)$ parallel time and uses $O(\log^3 n)$ states at each agent. Thereafter, a number of papers have been devoted to fast leader election, to name a few, [2, 14, 15, 19, 8]. Gąsieniec, Staehowiak, and Uznanski [15] gave an algorithm that converges in $O(\log n \log\log n)$ time and uses a surprisingly small number of states: only $O(\log\log n)$ states per agent. This is space-optimal because it is known that every leader election protocol with $O(n/\mathrm{polylog}(n))$ time uses $\Omega(\log\log n)$ states [1]. Sudo et al. [19] gave a simple protocol that elects a unique leader within $O(\log n)$ time and uses $O(\log n)$ states per agent. This is time-optimal because any leader election protocol requires $\Omega(\log n)$ time even if it uses an arbitrarily large number of states and the agents know the exact size of the population [17]. [3] At last, Berenbrink et al. [8] gave a time and space optimal protocol, i.e., an $O(\log n)$-time and $O(\log\log n)$-states leader election protocol.

SS-LE and LS-LE protocols have been presented also for a population where some pairs of agents may not have interactions, i.e., the interaction graph is not complete [11, 12, 20, 21, 23, 24].

2 Preliminaries

2.1 Model

We denote the set of integers $\{z \in \mathbb{N} \mid x \le z \le y\}$ by $[x, y]$. The omitted bases of logarithms are 2.

A *population* is the set V of n *agents* (i.e., $|V| = \mathrm{n}$) that change their states by pairwise interactions. Every pair of agents $(u, v) \in E = V \times V \setminus \{(w, w) \mid w \in V\}$ can interact with each other. A *protocol* P on the population is defined by a 4-tuple $P = (Q, Y, T, \pi_{out})$ consisting of a finite set Q of states, a finite set Y of output symbols, a transition function $T : Q \times Q \to Q \times Q$, and an output function $\pi_{out} : Q \to Y$. When two agents interact, T determines their next states based on their current states. The output function π_{out} maps the current local state $q \in Q$ to a value in the output domain $\pi_{out}(q) \in Y$. The state of each agent including the current output are often described as a set of local variables. Throughout this paper, we use the notation $v.x$ to denote the value of a variable x managed by agent v.

We assume that all agents have a common knowledge N on n such that $n \le N = O(n^c)$ holds for some constant c, which is equivalent to the assumption that the agents have a constant-factor approximation m of $\log n$, i.e., $\alpha \log n \ge m \ge \log n$ for some constant $\alpha \ge 1$[4].

A *configuration* is a mapping $C : V \to Q$ that specifies the states of all agents. Given a protocol P on n agents, the set of all possible configurations for P is denoted by $\mathcal{C}_{\mathrm{all}}(P)$. We say that a configuration C changes to C' by an interaction $e = (u, v)$, denoted by $C \overset{P,e}{\to} C'$, if $(C'(u), C'(v)) = T(C(u), C(v))$ and $C'(w) = C(w)$ for all $w \in V \setminus \{u, v\}$. Then u and v are respectively called the *initiator* and the *responder* of e. Given an interaction e, we say that agent $v \in V$ *participates* in e if v is either the initiator or the responder of e.

We assume the *uniformly random scheduler* $\mathbf{\Gamma}$, which selects two agents to interact at each step uniformly at random from all pairs of agents. Specifically, $\mathbf{\Gamma} = \Gamma_0, \Gamma_1, \dots$ where each $\Gamma_t \in E$ is a random variable such that $\Pr(\Gamma_t = (u, v)) = \frac{1}{n(n-1)}$ for any $t \ge 0$ and any distinct $u, v \in V$. Given an initial configuration $C_0 \in \mathcal{C}_{\mathrm{all}}(P)$, the *execution* of protocol P under the uniformly random scheduler $\mathbf{\Gamma}$ is defined as $\Xi_P(C_0, \mathbf{\Gamma}) = C_0, C_1, \dots$ where $C_t \overset{P,\Gamma_t}{\to} C_{t+1}$ holds for all $t \ge 0$. Note that each C_i is also a random variable.

[3] This lower bound may look obvious, but it is not: it does not immediately follow from a simple coupon collector argument because unlike SS-LE/LS-LE setting, we can now specify an initial configuration such that all agents are followers.

[4] In this sense, any protocol P should be parametric (with respect to m) such as $P_m = (Q_m, Y_m, T_m, \pi_{\mathrm{out},m})$ strictly. In this paper, we do not explicitly state parameter m of P for simplicity.

2.2 Loosely-Stabilizing Leader Election

In leader election protocols, every agent is equipped with an output variable $\texttt{leader} \in \{0, 1\}$, which indicates whether the agent is a leader. That is, if $v.\texttt{leader} = 1$ holds, v is a leader, and a follower otherwise. A configuration C is called *correct with leader* $v \in V$ if v outputs 1 and all other agents output 0. Given any configuration C, we define $\mathrm{EIH}_P(C)$ as the expected length of the longest prefix of $\Xi_P(C, \mathbf{\Gamma})$, where any configuration is correct with a common leader $v \in V$. Note that $\mathrm{EIH}_P(C) = 0$ holds if a configuration C is not correct. For any configuration C and any subset $\mathcal{S} \subseteq \mathcal{C}_{\mathrm{all}}(P)$ of configurations, we also define $\mathrm{EIC}_P(C, \mathcal{S})$ as the expected length of the longest prefix of $\Xi_P(C, \mathbf{\Gamma})$, where any configuration is not in $\mathcal{S}$. The notation EIH (resp. EIC) stands for the Expected number of Interactions to Hold (resp. Converge).

▶ **Definition 1** (Loosely-stabilizing leader election [18])**.** *Let α and β be positive real numbers. Protocol $P(Q, Y, T, \pi_{out})$ is an (α, β)-loosely-stabilizing leader election protocol if there exists a set $\mathcal{S}$ of configurations satisfying the two inequalities*

$$\max_{C \in \mathcal{C}_{\mathrm{all}}(P)} \mathrm{EIC}_P(C, \mathcal{S}) \leq \alpha \quad \textit{and} \quad \min_{C \in \mathcal{S}} \mathrm{EIH}_P(C) \geq \beta.$$

We call $\mathcal{S}$ defined by the definition above the set of *safe* configurations of P. Note that the condition $\beta > 0$ guarantees the correctness (i.e., uniqueness of leader) of configurations in $\mathcal{S}$. In terms of parallel time, an (α, β)-loosely-stabilizing leader election protocol P reaches a safe configuration within α/n parallel time in expectation, and it keeps the elected leader during the following β/n parallel time in expectation. We call α/n and β/n *the expected convergence time* and *the expected holding time* of P, respectively.

3 Toolbox

3.1 Epidemic

The protocol *epidemic* [6], denoted by P_{EP}, is often used to propagate the maximum value of a variable to the whole population, which is defined as: (i) each agent has only one variable x, and (ii) when two agents u and v interact, they substitute $\max(u.x, v.x)$ for their variables (i.e., $u.x$ and $v.x$). Then, we have the following lemma.

▶ **Lemma 2** ([6])**.**[5] *Let k be any non-negative integer, $D_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{EP}})$ be any configuration of P_{EP}, and $l = \max_{v \in V} v.x$ in configuration D_0. The execution $\Xi_{P_{\mathrm{EP}}}(D_0, \mathbf{\Gamma})$ reaches the configuration such that $u.x = l$ holds for any $u \in V$ within $O(kn \log n)$ steps with probability $1 - O(n^{-k})$.*

3.2 Countdown with Higher Value Propagation

The protocol of *counting down with higher value propagation* (CHVP) [18] is a useful technique to design loosely-stabilizing protocols, particularly for detecting the absence of a leader. It is defined as the following protocol P_{CD}: each agent has only one variable y, and when two agents u and v interact, they substitute $\max(u.y - 1, v.y - 1, 0)$ for their y. We have the following two lemmas.

[5] While the original protocol by Angluin et al. [6] is an one-way version of P_{EP} (i.e., higher value is propagated only from an initiator to a responder), there is no difference on asymptotic propagation time between them (Lemma 8 in [18]).

▶ **Lemma 3** (Lemma 1 in [3]).[6] *Let l_1 and l_2 be any two integers such that $l_1 > l_2 \geq 0$, k be any non-negative integer, and $D_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{CD}})$ be any configuration of P_{CD} such that $l_1 = \max_{v \in V} v.y$ holds. The execution $\Xi_P(D_0, \mathbf{\Gamma})$ reaches a configuration satisfying $\max_{v \in V} v.y \leq l_2$ within $O(n(l_1 - l_2 + k \log n))$ steps with probability $1 - O(n^{-k})$.*

▶ **Lemma 4** (Lemma 5 in [22]).[7] *Let $D_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{CD}})$ be any configuration and l be the integer that satisfies $l = \max_{v \in V} v.y$ at D_0. There exists a constant c such that $\Xi_{P_{\mathrm{CD}}}(D_0, \mathbf{\Gamma})$ reaches a configuration satisfying $\min_{v \in V} v.y \geq l - ck \log n$ within $O(kn \log n)$ steps with probability $1 - O(n^{-k})$.*

3.3 Lottery Game and Quick Elimination

The *lottery game*, originally introduced by Alistarh et al. [1] as a part of their leader election protocol, is a probabilistic process of filtering leaders. An abstract form of the lottery game is stated as follows: Let V' be the set of leaders. Every leader $v \in V'$ makes independent fair coin flips until it observes tail for the first time. Then, the number of observed heads s_v (called the *level* of v) is propagated to other leaders. The agent identifying another agent with a higher level drops out as a loser.

There are a few implementations of the lottery game in population protocol models. Alistarh et al. [1] and Sudo et al. [19] develop (non-loosely-stabilizing) leader election protocols, based on their own implementations and analyses for this game. In this paper, we adopt the implementation shown in [19], called *quick elimination* (QE). The pseudocode of QE is given in Algorithm 1, which describes the state transition when two agents a_0 and a_1 interact, where a_0 is an initiator and a_1 is a responder. Since the propagation of level values is easily implemented by the epidemic, the main non-trivial point is how to synthesize coin flips using the randomness of the scheduler. The implementation QE simply utilizes the asymmetry of interactions. That is, if v joins an interaction as the initiator, it receives head as the result of its coin flip, and receives tail if it joins as the responder. Each agent maintains two variables, `done` and `level`, in addition to an output variable `leader`. The flag `done` $\in \{0, 1\}$ implies whether the agent is still in the decision of its level (i.e., it continues (synthetic) coin flips during `done` $= 0$). Starting from the state with `done` $= 0$ and `level` $= 0$, the agent v with v.`leader` $= 1$ first decides its level: it increments v.`level` every time it observes head, and it stops incrementation and sets `done` to 1 when it observes tail for the first time. Agents that have decided their levels perform the epidemic to share the maximum level (lines 6-7). If an agent sees a higher level, it becomes a follower (line 7).

While the lottery game was used as a scheme to eliminate leaders in the past literature, we rather see it as a Monte Carlo protocol for leader election, i.e., we focus on the probability that exactly one player wins (or survives as a leader). The following lemma is the key ingredient of our protocol, which is simple but a new observation that has not been addressed so far.[8]

[6] Precisely, k is assumed to be a constant in the original lemma, but the same proof applies in the case that k depends on n.

[7] We obtain this lemma by substituting $d = k + 3$, $d' = k + 3$, $d'' = 6$, and $t = \lceil kn \ln n \rceil$ for the first inequality in Lemma 5 in [22].

[8] This observation might not be new if the results of coin flips by the agents were independent of each other. However, they are not independent in our model because when two agents have an interaction, one of them observes head and the other observes tail. Thus, to the best of our knowledge, this observation is new.

Algorithm 1 $\boldsymbol{QE}()$ (a_0 is an initiator and a_1 is a responder).

```
for $i \in \{0,1\}$ s.t. $a_i$.done $= 0 \wedge a_i$.leader $= 1$ do
    if $i = 0$ then
        $a_0$.level $\leftarrow \min(a_0$.level $+ 1, 2m)$
    else
        $a_1$.done $\leftarrow 1$
if ($a_0$.done $= 1 \wedge a_1$.done $= 1$
    $\wedge\, \exists i \in \{0,1\} : a_i$.level $< a_{1-i}$.level) then
    $a_i$.leader $\leftarrow 0$;   $a_i$.level $\leftarrow a_{1-i}$.level
```

▶ **Lemma 5.** *Consider the execution of QE under the uniformly random scheduler* $\mathbf{\Gamma}$ *starting from a configuration where at least one leader exists and* `done` $= 0$ *and* `level` $= 0$ *hold for all agents. When all leaders finish deciding their levels (i.e.,* `done` $= 1$ *holds for all leaders), exactly one leader has the maximum level* ($\max_{v \in V} v.\texttt{level}$) *with probability at least* $1/16$.

Proof. Let V' be the set of leaders at the initial configuration. Let X_v be the level computed by agent $v \in V'$. That is, X_v is the integer such that $v \in V'$ joins X_v interactions as an initiator before it joins an interaction as a responder for the first time. We show that $p_u = \Pr(X_u \geq \lceil \log n \rceil + 2 \wedge \bigwedge_{v \in V' \setminus \{u\}} X_v \leq \lceil \log n \rceil + 1) \geq 1/16n$ holds for any $u \in V'$. Then, the probability that some agent becomes the unique winner is obviously lower bounded by $\sum_{u \in V'} p_u \geq 1/16$. Thus, the lemma holds because when `done` $= 1$ holds for all leaders, every agent except for the unique winner has a smaller level or must have become a follower before. Since $\Pr(X_u \geq \lceil \log n \rceil + 2) > 1/8n$ holds, it suffices to show $q = \Pr(\bigwedge_{v \in V' \setminus \{u\}} X_v \leq \lceil \log n \rceil + 1 \mid X_u \geq \lceil \log n \rceil + 2) \geq 1/2$. By the union bound, we have $q \geq 1 - \sum_{v \in V' \setminus \{u\}} \Pr(X_v \geq \lceil \log n \rceil + 2 \mid X_u \geq \lceil \log n \rceil + 2)$. When two agents u and v both with `done` $= 0$ interact with each other, one of them necessarily reaches the decision of its level. That is, u and v have at most one common interaction before either one decides its level. This implies that to obtain $X_v \geq \lceil \log n \rceil + 2$ under the condition $X_u \geq \lceil \log n \rceil + 2$, v must observe at least $\lceil \log n \rceil + 1$ heads at the coin flips independently of the first $\lceil \log n \rceil + 2$ coin flips by u. That is, we have $\Pr(X_v \geq \lceil \log n \rceil + 2 \mid X_u \geq \lceil \log n \rceil + 2) \leq (1/2)^{\lceil \log n \rceil + 1} \leq 1/2n$, and thus, $q \geq 1 - n \cdot (1/2n) \geq 1/2$. ◀

4 Time-optimal LS-LE

In this section, we give an LS-LE protocol $P_{\mathrm{TO}}(\tau)$, where the integer $\tau \geq 1$ is a design parameter controlling the performance of the protocol. Starting from any initial configuration, this protocol reaches a safe configuration within $O(\tau n \log n)$ steps and keeps the single leader in the following $\Omega(n^\tau)$ steps. The number of states per agent is $\Theta(\tau m) = \Theta(\tau \log n)$. In the rest of this paper, we use terminologies "*with high probability*" to mean "with probability $1 - O(1/n)$" and "*with very high probability*" to mean "with probability $1 - O(1/n^\tau)$". Further, the terminology "quickly" is used for implying "within $O(\tau n \log n)$ steps".

4.1 Protocol in a Nutshell

The protocol $P_{\mathrm{TO}}(\tau)$ elects a unique leader by iteratively performing the following two phases, both taking $\Theta(\tau n \log n)$ steps with very high probability.

- **Check phase**: The protocol checks whether the population has at least one leader. Each leader agent propagates a heartbeat message to all others using the epidemics. The agents not receiving that message until the end of the phase conclude that the population has no leader, and they become leaders. Since two or more agents may become leaders, they are filtered in the election phase.
- **Election phase**: Each agent performs QE. As shown in Lemma 5, this phase decreases the number of leader agents to one with a constant probability.

There are two major issues for implementing these phases: how to realize a loosely-stabilizing synchronization mechanism to yield the transition between two phases, and how to combine it with the task of each phase using only a small number of states. The protocol CHVP, stated in Section 3.2, is one of the possible solutions for the first issue, which provides a loosely-stabilizing (synchronized) timeout mechanism; thus, it can be utilized for global phase synchronization. However, addressing the second issue is more challenging. Since the check phase only consumes a constant number of states, it is easily combined with CHVP. In the election phase, both QE and CHVP internally keep a variable of a non-constant size. The former manages a variable `level` $\in [0, 2m]$, and the latter manages a variable whose range is $[0, O(\tau m)]$, as we will see in Section 4.2. Thus, to bound the number of states by $O(\tau m) = O(\tau \log n)$ in total, they must share a single non-constant variable.

We resolve this matter by designing a new loosely-stabilizing task sharing scheme called *mode switching*. Unlike the task-sharing techniques in the past literature [15, 19], it *dynamically* changes the mode of each agent during the election phase. The two modes respectively correspond to synchronization and QE, and each agent is engaged in the task associated with its own mode. In total, the protocol is equipped with three different roles of agents, i.e., check phase, synchronization in election phase, and QE in election phase. We call each role a *class* of agents, and they are respectively referred to as *checker*, *synchronizer*, and *elector*. It should be noted that dynamic mode change is crucial for attaining loose stabilization: A non-correct initial configuration filled by electors obviously causes a deadlock because the timeout of the election phase never occurs. Thus, it is indispensable to install a mechanism that changes electors to synchronizers. That mechanism, however, prevents the quick propagation of the maximum level in QE owing to the lack of a sufficiently large number of electors (recall that even agents not involved in the lottery game must work as a medium in the epidemic). In fact, if only $o(n)$ electors remain, we cannot guarantee with very high probability that the epidemic of the maximum level finishes quickly. This observation implies that the mode change from synchronizers to electors is also necessary.

The remaining concern is how to design synchronization and QE with adapting to dynamic change. The task of QE is robust for such dynamics if an agent with mode change always joins as a follower. However, CHVP is not robust because the countdown timer is rewound by a newly joining agent with a high counter value. Fortunately, we can obtain an alternative solution for this matter: simply using a local countdown timer, which just counts the number of interactions performed by the timer holder. While CHVP is necessary to recover global synchronization from the highly deviated situations where two agents are in different phases or have two counter values with a large difference, we can delegate such a role entirely to the check phase. Then, the election phase can use the timeout mechanism not necessarily synchronized among all agents.

4.2 Variables and Groups

For describing the protocol, we use two (hard-coded) fixed values, $r_{\max}$ and $b_{\max}$, both of which are $\Theta(\tau m) = \Theta(\tau \log n)$ for sufficiently large hidden constants. We also define $r_{\mathrm{mid}} = cr_{\max}$ for an appropriate $1 > c > 0$ such that $c/(1-c)$ becomes sufficiently large. All

Table 2 Variables used in protocol P_{TO}. Each time an agent changes its class, class-specific variables are set to the initial value specified below. The initial value of a variable `detect` is not a fixed value: the value of a common variable v.`leader` is copied to v.`detect` each time an agent v becomes a checker.

	Variable name	Initial value
Common variables	$\texttt{leader} \in \{0,1\}$	-
	$\texttt{phase} \in \{CH, EL\}$	-
	$\texttt{mode} \in \{A, B\}$	-
Variables for checkers	$\texttt{timer}_R \in [0, r_{\max}]$	$r_{\max}$
	$\texttt{detect} \in \{0,1\}$	`leader`
Variables for electors	$\texttt{level} \in [0, 2m]$	0
	$\texttt{done} \in \{0,1\}$	0
Variables for synchronizers	$\texttt{timer}_B \in [0, b_{\max}]$	$b_{\max}$

Table 3 Descriptors for specific subsets of agents.

$V_L = \{v \in V \mid v.\texttt{leader} = 1\}$, $V_F = \{v \in V \mid v.\texttt{leader} = 0\}$
$V_{CH} = \{v \in V \mid v.\texttt{phase} = CH\}$, $V_{EL} = \{v \in V \mid v.\texttt{phase} = EL\}$
$V_A = \{v \in V_{EL} \mid v.\texttt{mode} = A\}$, $V_B = \{v \in V_{EL} \mid v.\texttt{mode} = B\}$
$V_{CH\geq} = \{v \in V_{CH} \mid r_{\mathrm{mid}} \leq v.\texttt{timer}_R \leq r_{\max}\}$
$V_{CH<} = \{v \in V_{CH} \mid 0 \leq v.\texttt{timer}_R < r_{\mathrm{mid}}\}$
$V_{\mathrm{done}} = \{v \in V_L \cap V_A \mid v.\texttt{done} = 1\}$
$V_{\mathrm{undone}} = \{v \in V_L \cap V_A \mid v.\texttt{done} = 0\}$

the hidden constants are appropriately fixed in the "on-demand" manner in the proof details. We also assume $n \geq 3$ for the simplicity of argument; however, it is not essential. It can be easily observed that this protocol is a self-stabilizing leader election protocol in the case of $n = 2$.

The set of variables used in protocol P_{TO} is shown in Table 2. As stated in Section 4.1, in protocol P_{TO}, there are three classes of agents: checkers, synchronizers, and electors. Each class has a set of variables specific for the associated task, and an agent manages the variables related to its own class as well as the set of common variables. Note that the list of variables in Table 2 contains two $\Theta(\tau \log n)$-state variables ($\texttt{timer}_R$ and $\texttt{timer}_B$) and one $\Theta(\log n)$-state variable (`level`), but they are used exclusively. That is, at any configuration, each agent has the responsibility of managing only one of the three. Thus, the total number of states necessary for storing all variables in Table 2 is bounded by $O(\tau \log n)$. The column "Initial value" in Table 2 indicates the initial values set for class-specific variables. The initialization occurs when the agent changes its class. For avoiding unnecessary complication, this initialization process is not explicitly stated in the pseudocode presented later. The class of each agent is identified by two common variables `phase` and `mode`. More precisely, the agent v with $v.\texttt{phase} = CH$ is a checker, that with $v.\texttt{phase} = EL$ and $v.\texttt{mode} = A$ an elector, and that with $v.\texttt{phase} = EL$ and $v.\texttt{mode} = B$ a synchronizer. The set of agents belonging to each class is denoted by V_{CH}, V_A, and V_B respectively. In addition, we introduce several notations for describing the set of agents satisfying some condition, as listed in Table 3.

4.3 Details of the Protocol

The pseudocode of P_{TO} is shown in Algorithm 2. The main bodies of the two phases are realized by lines 3 and 16 and the procedure *GoToElection*(). Line 3, which corresponds to the check phase, performs the propagation of detect flags (i.e., the existence of leader

Algorithm 2 $\boldsymbol{P_{\mathrm{TO}}}$ (a_0 is an initiator and a_1 is a responder).

```
1  for each $i \in \{0,1\}$ s.t. $a_i \in V_B$ do $a_i.\mathtt{leader} \leftarrow 0$
2  if $a_0, a_1 \in V_{CH}$ then
3      $a_0.\mathtt{detect} \leftarrow a_1.\mathtt{detect} \leftarrow \max(a_0.\mathtt{detect}, a_1.\mathtt{detect})$
4      $a_0.\mathtt{timer}_R \leftarrow a_1.\mathtt{timer}_R \leftarrow \max(a_0.\mathtt{timer}_R - 1, a_1.\mathtt{timer}_R - 1, 0)$
5      if $a_0.\mathtt{timer}_R = 0$ then
6          $\mathit{GoToElection}(0)$; $\mathit{GoToElection}(1)$
7  else if $\exists i \in \{0,1\} : a_i \in V_{EL} \wedge a_{1-i} \in V_{CH\geq}$ then
8      $a_i.\mathtt{phase} \leftarrow \mathit{CH}$                    // Reset checker's variables by Table 2
9  else if $\exists i \in \{0,1\} : a_i \in V_{CH<} \wedge a_{1-i} \in V_{EL}$ then
10     $\mathit{GoToElection}(i)$
11 if $a_0, a_1 \in V_{EL}$ then
12     if $a_0, a_1 \in V_A \cap V_F \wedge a_0.\mathtt{level} = a_1.\mathtt{level}$ then
13         $a_1.\mathtt{mode} \leftarrow B$          // Reset synchronizer's variables by Table 2
14     else if $a_0, a_1 \in V_B$ then
15         $a_i.\mathtt{mode} \leftarrow A$ where $i = \max\{j \in \{0,1\} \mid a_j.\mathtt{timer}_B \geq a_{1-j}.\mathtt{timer}_B\}$
                                              // Reset elector's variables by Table 2
16     $\mathit{QE}()$
17     if $a_0, a_1 \in V_{\mathrm{done}} \wedge a_0.\mathtt{level} = a_1.\mathtt{level}$ then $a_1.\mathtt{leader} \leftarrow 0$      // $V_{\mathrm{done}} \subseteq V_L$
18     for each $i \in \{0,1\}$ s.t. $a_i \in V_B$ do $a_i.\mathtt{timer}_B \leftarrow \max(a_i.\mathtt{timer}_B - 1, 0)$
19     for each $i \in \{0,1\}$ s.t. $a_i \in V_B \wedge a_i.\mathtt{timer}_B = 0$ do $a_i.\mathtt{phase} \leftarrow \mathit{CH}$
                                              // Reset checker's variables by Table 2
20 function $\mathit{GoToElection}(i)$:
21     if $a_i.\mathtt{detect} = 0$ then $a_i.\mathtt{leader} \leftarrow 1$
22     $(a_i.\mathtt{phase}, a_i.\mathtt{mode}) \leftarrow (\mathit{EL}, A)$     // Reset elector's variables by Table 2
```

agents) using the epidemic. Line 16 indeed corresponds to the task of QE. The procedure *GoToElection*() corresponds to the phase transition from check to election, where the agent not detecting the existence of leaders becomes a leader. The remaining part is devoted to the synchronization mechanism including the mode switching scheme. Lines 4-6 correspond to the implementation of CHVP, where the timer variable $\mathtt{timer}_R$ is updated (line 4), and the transition to the election phase is triggered when timeout occurs (lines 5 and 6). Lines 7-10 are the mechanism supporting smooth phase transition, which is crucial for guaranteeing the correctness criteria of the synchronization mechanism explained later. Lines 7-8 and 9-10 respectively address the transition from check to election and its reversal. Lines 11-19 correspond to the task for synchronizers and electors. The core of this part is the mode switching scheme, described in lines 12-15. The switch from elector to synchronizer happens when a follower agent interacts with another follower with the same level (lines 12-13), and the opposite occurs when two synchronizers interact with each other (lines 14-15). It is shown in the next section that this scheme appropriately control the size of two classes. Line 18 is the countdown of local timers held by synchronizers, and line 19 is the phase transition from election to check. The leader elimination in line 17 is not for the leader election itself, but rather to handle the initial configurations consisting only of leaders with the same level. Without this code, the protocol would be deadlocked in that case. A synchronizer is always a follower, as guaranteed by line 1.

For stating the precise goal of the synchronization mechanism, we explain its intended behavior as well as the concise reason why such a behavior is attained.

Behavior 1 Starting from any configuration in $\mathcal{C}_{\text{all}}(P_{\text{TO}}(\tau))$, the population quickly reaches a configuration where $V = V_{CH\geq}$ holds with very high probability. The CHVP protocol shrinks the large deviation among all timers in the check phase. Therefore, once an agent goes back to the check phase from the election phase and resets its $\texttt{timer}_R$ to $r_{\max}$, the population quickly reaches a configuration where $V = V_{CH\geq}$. One may think that the population gets stuck in the election phase once it reaches a configuration where all agents are electors (i.e., there is no synchronizers). However, even starting from such a configuration, the population quickly creates at least one synchronizer because the following events occur with very high probability: (i) all leaders quickly decide their levels; (ii) the maximum level quickly propagates to the whole population as long as there is no synchronizer; (iii) since all the agents have the same level, the number of followers quickly becomes $\Theta(n)$ as long as there is no synchronizer; and (iv) two followers with the same level have an interaction quickly and one of them becomes a synchronizer. Once a synchronizer is created, some agent quickly goes back to the check phase because each synchronizer simply counts down its local timer.

Behavior 2 Once the current configuration satisfies $V = V_{CH\geq}$, CHVP decreases timer values (i.e., $\texttt{timer}_R$) while maintaining a relatively smaller deviation among agents. Since timer values of agents in $V_{CH\geq}$ are all $\Theta(\tau \log n)$, the check phase continues during $\Theta(\tau n \log n)$ steps with very high probability. When an agent is timed out, it moves to the election phase. Then, owing to the low deviation of CHVP timers, no agent is still in $V_{CH\geq}$, and thus, the transition in lines 9-10 quickly takes all other agents to the election phase with very high probability. During this period, no agent goes back to the check phase from the election phase with very high probability because the upper limit $b_{\max}$ of $\texttt{timer}_B$ is $\Theta(\tau \log n)$ with a sufficiently large hidden constant.

Behavior 3 In the election phase, the fastest timer (i.e., the agent with the smallest timer value) of all synchronizers determines the pace. Since it is never rewound, the election phase keeps $\Theta(\tau n \log n)$ steps with very high probability. Similar to the behavior from check to election, when an agent becomes a checker, all other agents are quickly brought back to the check phase with very high probability. During this period, no agent goes to the election phase from the check phase with very high probability because the upper limit $r_{\max}$ of $\texttt{timer}_R$ is $\Theta(\tau \log n)$ with a sufficiently large hidden constant.

The correctness criteria of the synchronization mechanism is that the system iterates Behaviors 2 and 3 with very high probability after recovery from unintended situations (by Behavior 1), which is necessary for our protocol to elect a unique leader in the loosely-stabilizing manner. The formal proof of the correctness is given in the next section.

In $QE()$, we expect that the largest level is quickly propagated to all leaders with very high probability. Sudo et al. [19] proved that this is true if $|V_A| = \Theta(n)$ holds and V_A remains the same during this period. However, P_{TO} frequently executes the mode switching from A to B and from B to A. Without the mode switching, the number of agents with $\texttt{level} = s_{\max}$ is monotonically non-decreasing, while with the mode switching, it decreases when an agent with $\texttt{level} = s_{\max}$ changes its mode from A to B. Therefore, we must evaluate the effect of the mode switching on the speed of the propagation. Fortunately, there is no severe effect of the mode switching for our purpose: every leader in V_{done} whose $\texttt{level}$ is not the largest becomes a follower within $O(n \log n)$ steps with probability $1 - o(1)$.

5 Analysis

To express claims in a formal manner, we first define the following notations.

- $\mathcal{A}_X$: the set of all configurations $\mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))$ where $V_X = V$ holds. For example, $\mathcal{A}_{CH}$ is the set of all configurations where every agent is in the check phase (i.e., $V_{CH} = V$).
- $\mathcal{C}_{EL\geq}$: the set of all configurations in $\mathcal{A}_{EL}$ where $v.\mathtt{timer}_B \geq b_{\max}/2$ holds for every $v \in V_B$.
- $\mathcal{C}_{\mathrm{reset}}$: the set of all configurations in $\mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))$ where there is at least one agent $v \in V_{CH}$ such that $v.\mathtt{timer}_R = r_{\max}$.

The first goal of this section is to prove the following three lemmas (Lemmas 6, 7, 8). Intuitively, Lemma 6 claims that synchronization is recovered quickly with very high probability from any configuration in $\mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))$, and Lemmas 7 and 8 claim that once the synchronization is recovered, the check phase and the election phase are iterated thereafter, both taking $\Theta(\tau n \log n)$ steps with sufficiently large hidden constants, with very high probability.

In the rest of this paper, for any set $\mathcal{C}$, we say that an execution *enters* $\mathcal{C}$ when it reaches a configuration in $\mathcal{C}$.

▶ **Lemma 6.** *Let C_0 be any configuration in $\mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))$ and let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. Execution Ξ enters $\mathcal{A}_{CH\geq}$ quickly with very high probability.*

▶ **Lemma 7.** *Let C_0 be any configuration in $\mathcal{A}_{CH\geq}$ and let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. Then, the following hold with very high probability:*

1. *execution Ξ enters $\mathcal{C}_{EL\geq}$ quickly,*
2. *no agent moves from the election phase to the check phase before Ξ enters $\mathcal{C}_{EL\geq}$, and*
3. *execution Ξ stays in $\mathcal{A}_{CH}$ for $\Omega(nr_{\mathrm{mid}}) = \Omega(\tau n \log n)$ steps.*

▶ **Lemma 8.** *Let C_0 be any configuration in $\mathcal{C}_{EL\geq}$ and let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. Then, the following hold with very high probability:*

1. *execution Ξ enters $\mathcal{A}_{CH\geq}$ quickly,*
2. *no agent moves from the check phase to the election phase before Ξ enters $\mathcal{A}_{CH\geq}$, and*
3. *execution Ξ stays in $\mathcal{A}_{EL}$ for $\Omega(nr_{\mathrm{mid}}) = \Omega(\tau n \log n)$ steps.*

In what follows, we first prove Lemma 6 by giving four supplemental lemmas (Lemmas 9, 10, 11, and 12). We next prove Lemmas 7 and 8.

▶ **Lemma 9.** *Starting from any configuration $C_0 \in \mathcal{A}_{EL}$, execution $\Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$ quickly enters $\mathcal{C}_{\mathrm{reset}}$ or reaches a configuration in $\mathcal{A}_{EL}$ satisfying $V_{\mathrm{undone}} = \emptyset$ with very high probability.*

Proof. Let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. When an agent goes back to the check phase from the election phase, it substitutes $r_{\max}$ for its $\mathtt{timer}_R$ (lines 8 and 19). Therefore, Ξ never leaves $\mathcal{A}_{EL}$ until it enters $\mathcal{C}_{\mathrm{reset}}$. Let v be any agent that satisfies $v \in V_{\mathrm{undone}}$ in C_0. As long as $v \in V_{\mathrm{undone}}$ holds, v makes a coin flip every time v has an interaction. Since every agent joins an interaction with probability $2/n$ at each step, by the Chernoff bound, v has $2 \log n$ or more interactions within sufficiently large $O(n \log n)$ steps with probability $1 - O(1/n^2)$. Therefore, $v.\mathtt{done} = 1$ holds within $O(n \log n)$ steps with probability $1 - (1/2)^{2 \log n} - O(1/n^2) = 1 - O(1/n^2)$ because each coin flip results in "tail" with probability exactly $1/2$, by which v leaves V_{undone}. By the union bound, execution Ξ enters $\mathcal{C}_{\mathrm{reset}}$ or reaches a configuration in $\mathcal{A}_{EL}$ satisfying $V_{\mathrm{undone}} = \emptyset$ within $O(n \log n)$ steps with probability $1 - O(1/n)$. We obtain the lemma by repeating this analysis τ times. ◀

▶ **Lemma 10.** *Starting from any configuration $C_0 \in \mathcal{A}_{EL}$ satisfying $V_{\mathrm{undone}} = \emptyset$, execution $\Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$ quickly reaches a configuration in $\mathcal{A}_{EL}$ satisfying $V_B \neq \emptyset$ with very high probability.*

Proof. Let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. In execution Ξ, no leader has $\mathtt{done} = 0$ before Ξ reaches a configuration in $\mathcal{A}_{EL}$ where $V_B \neq \emptyset$. Therefore, it suffices to show that Ξ reaches a configuration in $\mathcal{A}_{EL}$ where $V_B \neq \emptyset$ within $O(n \log n)$ steps with high probability because we obtain the lemma by repeating this trial τ times.

While both $V_{\mathrm{undone}} = \emptyset$ and $V_B = \emptyset$ hold, nothing prevents the epidemic from propagating the maximum value of $\mathtt{levels}$. Thus, by Lemma 2, the maximum value is propagated to the whole population within $O(n \log n)$ steps with high probability [6]. Once all agents have the same level, the number of followers increases by one every time two leaders meet (line 17). As long as $|V_F| < n/2$ and $V_B = \emptyset$ hold, two leaders meet each other with probability at least $1/4$ at each step. Hence, $|V_F| \geq n/2$ or $V_B \neq \emptyset$ holds within $O(n \log n)$ steps with high probability. In the former case, at each step thereafter, two followers have an interaction and one of them becomes a synchronizer (line 13) with a constant probability. Therefore, $|V_B| \neq \emptyset$ holds within $O(\log n)$ steps with high probability. ◀

▶ **Lemma 11.** *Starting from any configuration $C_0 \in \mathcal{A}_{EL}$ satisfying $V_B \neq \emptyset$ holds, execution $\Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$ quickly enters $\mathcal{C}_{\mathrm{reset}}$ with very high probability.*

Proof. Before $\Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$ enters $\mathcal{C}_{\mathrm{reset}}$, the smallest $\mathtt{timer}_B$ in the population (i.e., $\min_{v \in V_B} \mathtt{timer}_B$) is monotonically non-increasing. It decreases by one or a timeout of $\mathtt{timer}_B$ occurs when a synchronizer with the smallest $\mathtt{timer}_B$ has an interaction (line 18), which occurs with probability at least $2/n$ at each step. Since $b_{\max} = \Theta(\tau \log n)$, by the Chernoff bound, some synchronizer encounters the timeout of $\mathtt{timer}_B$ quickly with very high probability. ◀

▶ **Lemma 12.** *Starting from any configuration $C_0 \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))$, execution $\Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$ quickly enters $\mathcal{C}_{\mathrm{reset}}$ with very high probability.*

Proof. Let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. Note that Ξ enters $\mathcal{C}_{\mathrm{reset}}$ whenever an agent goes back to the check phase from the election phase. Before Ξ enters $\mathcal{C}_{\mathrm{reset}}$ or $\mathcal{A}_{EL}$, $\max_{v \in V_{CH}} v.\mathtt{timer}_R$ is monotonically non-increasing and this value decreases at a pace faster than or equal to the pace at which the maximum value of variable y decreases in the CHVP protocol in Section 3. Therefore, by Lemma 3 with $l_1 = r_{\max}$, $l_2 = 0$, $k = \tau$, Ξ reaches a configuration $C' \in \mathcal{C}_{\mathrm{reset}} \cup \mathcal{A}_{EL}$ quickly with very high probability. Once Ξ enters $\mathcal{A}_{EL}$, it enters in $\mathcal{C}_{\mathrm{reset}}$ quickly with very high probability by Lemmas 9, 10, and 11. ◀

Proof of Lemma 6. By Lemma 12, we can assume that $C_0 \in \mathcal{C}_{\mathrm{reset}}$. Since we assume $r_{\max} - r_{\mathrm{mid}} = O(\tau \log n)$ with a sufficiently large hidden constant, by Lemma 4 with $l = r_{\max}$ and $k = \tau$, execution $\Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$ enters $\mathcal{A}_{CH\geq}$ quickly with very high probability. ◀

Proof of Lemma 7. The last claim is trivial because each agent has an interaction with probability $2/n$ at each step and at least one agent must have r_{mid} interactions until execution Ξ leaves $\mathcal{A}_{CH}$. We prove the first and the second claims below.

By Lemma 3 with $k = \tau$, Ξ enters $\mathcal{A}_{CH<}$ within $O(n(r_{\max} - r_{\mathrm{mid}})) = O(\tau n \log n)$ steps or at least one agent goes to the election phase during the period with very high probability. Since we assume that $r_{\mathrm{mid}}/(r_{\max} - r_{\mathrm{mid}})$ is a sufficiently large constant, together with the third claim and the union bound, we observe that Ξ enters $\mathcal{A}_{CH<}$ quickly with very high probability. Thereafter, by Lemma 3 with $k = \tau$, at least one agent goes to the election phase within $O(\tau n \log n)$ steps with very high probability. Remember that whenever an agent in $V_{CH<}$ and an agent in V_{EL} meet, the former moves to the election phase. Hence, once an agent goes to the election phase, the agents in the population go to the election phase one after another in completely the same way as the epidemic protocol in Section 3.1.

Therefore, by Lemma 2 with $k = \tau$, Ξ reaches a configuration $C \in \mathcal{A}_{EL}$ quickly with very high probability. Since $b_{\max}$ is sufficiently large, no agent has more than $b_{\max}/2$ interactions during this period with very high probability. Therefore, $C \in \mathcal{C}_{EL\geq}$ holds with very high probability. From the above, we conclude that the first and second claims also hold. ◀

Proof of Lemma 8. The last claim is trivial because each agent has an interaction with probability $2/n$ at each step and at least one agent must have $b_{\max}/2$ interactions until execution Ξ leaves $\mathcal{A}_{EL}$. By Lemmas 9, 10, and 11, Ξ reaches a configuration $\mathcal{C}_{\text{reset}}$ within $O(\tau n \log n)$ steps with very high probability. Thereafter, in completely the same way as given in the second paragraph of the proof of Lemma 7, we can prove that Ξ enters $\mathcal{A}_{CH\geq}$ quickly (by the epidemic) and $V_{CH<} = \emptyset$ always holds during this period with very high probability. Thus, the first claim holds. No agent goes to the election phase when it belongs to $V_{CH\geq}$, from which the second claim follows. ◀

▶ **Lemma 13.** *Let C_0 be a configuration in $\mathcal{A}_{EL}$ where $V_{\text{undone}} = \emptyset$ holds and let $\Xi = \Xi_{P_{\text{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. Let $V' \subset V_L \cap V_A$ be the set of leaders whose `level` are not the largest in C_0. Then, execution Ξ reaches a configuration in $\mathcal{A}_{EL}$ where all agents in V' are followers or enters $\mathcal{C}_{\text{reset}}$ within $O(n \log n)$ steps with probability $1 - o(1)$.*

Proof. In this proof, we ignore the case that some agent goes back to the check phase (i.e., Ξ enters $\mathcal{C}_{\text{reset}}$) because this ignorance only decreases the probability claimed in the lemma. Let l be the maximum level of the population (i.e., $\max_{v \in V_A} v.\texttt{level}$) in C_0. To obtain the lemma, it suffices to show that all leaders in V' observe the maximum level l and become followers within $O(n \log n)$ steps with probability $1 - o(1)$.

First, we analyze $n_A = |V_A|$ in execution Ξ. This value increases by one if two agents in V_B meet, and it decreases by one if two followers with the same level in V_A meet. Therefore, at each step where $n_A \leq n/3$, n_A increases with probability at least $4/9$, while n_A decreases with probability at most $1/9$. The gap of these probabilities and the Chernoff bound guarantee that even if $n_A < n/3$ in C_0, n_A reaches $n/3$ within $O(n)$ steps with high probability. Let C' be the configuration at this time. Once Ξ reaches C', the above gap of probabilities, the Chernoff bound, and the union bound guarantee that $n_A \geq n/4$ always holds for arbitrarily large $\Omega(n \log n)$ steps with high probability. Thus, we can assume $n_A \geq n/4$ in the following discussion on execution Ξ after C'.

Consider the suffix of Ξ after C'. Let $n_M = |\{v \in V_A \mid v.\texttt{level} = l\}|$. This value increases by one if an interaction happens between two agents in V_A such that one has the maximum level l and the other has a lower level. It decreases by one if an interaction happens between two followers in V_A, both with level l. Note that $n_M \geq 1$ always holds because n_M decreases only if two agents with level l have an interaction. Since we assume $n_A \geq n/4$, at each step where $n_M \leq n/8$, n_M increases with probability $p_{\text{inc}} \geq (n_M \cdot (n/4 - n_M))/\binom{n}{2} \geq n_M/(4n)$, while n_M decreases with probability $p_{\text{dec}} \leq n_M^2/n^2$. As long as $n/2^9 \leq n_M \leq n/2^8$, we have $p_{\text{inc}} \geq 1/2^{11}$ and $p_{\text{dec}} \leq 1/2^{16}$. This large difference between p_{inc} and p_{dec} guarantees that once n_M reaches $n/2^8$, $n_M \geq n/2^9$ always holds for arbitrarily large $\Omega(n \log n)$ steps with high probability, by the Chernoff bound and the union bound. During this period, each leader in V' meets an agent in V_A with the maximum level l with probability $\Omega(1/n)$ at each step. Thus, once $n_A \geq n/2^8$ holds, all leaders in V' become followers within $O(n \log n)$ steps with high probability.

Thus, all we have to do is to show that $n_M \geq n/2^8$ holds within $O(n \log n)$ steps starting from C'. First, we show that n_M reaches $24 \ln n$ or larger within $O(n \log n)$ steps starting from C'. When $n_M < 24 \ln n$, $p_{\text{inc}} = \Omega(1/n)$ and $p_{\text{dec}} = O((\log^2 n)/n^2)$ always hold. Therefore, by the Chernoff bound, n_M reaches $24 \ln n$ or larger within $O(n \log n)$ steps with high probability.

Next, for any integer k such that $24 \ln n \le k < n/2^8$, we show that once n_M reaches k, n_M reaches $2k$ with high probability. As long as $k/2 \le n_M \le 2k$, we have $p_{\mathrm{inc}} \ge k/8n$ and $p_{\mathrm{dec}} \le 4k^2/n^2 < k/64n$. Therefore, by the Chernoff bound, we have the followings:

- during the first $16n$ steps, n_M is always $k/2$ or larger with probability at least $1 - e^{-(1/3)\cdot(k/4)} = 1 - O(1/n^2)$,
- during the first $x \ge 16n$ steps, n_M increases at least $xk/16n$ times with probability $1 - O(1/n^2)$, and
- during the first $x \ge 16n$ steps, n_M decreases at most $xk/32n$ times with probability $1 - O(1/n^2)$.

Therefore, by the union bound (for $x = 16n, 16n+1, \dots, 32n$), n_M reaches $k + (2k - k) = 2k$ within $32n$ steps with high probability. Therefore, once n_M reaches $24 \ln n$ or larger value, it doubles in every $32n$ steps with high probability until it reaches $n/2^8$. Thus, n_M reaches $n/2^8$ within $O(n \log n)$ steps with probability $1 - O((\log n)/n) = 1 - o(1)$. ◀

By the correctness of the synchronization and Lemma 13, we can easily show Lemmas 15 and 16 (see Appendix for a complete proof), where we define a set $\mathcal{S}$ of safe configurations by Definition 14.

▶ **Definition 14** (Safe configurations). *Define $\mathcal{S}$ as the set of all configurations where $V = V_{CH\ge}$ holds, exactly one leader v_l exists in the population, and $v_l.\mathtt{detect} = 1$ holds.*

▶ **Lemma 15.** $\min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{TO}}(\tau)}(C) = \Omega(n^{\tau+1})$.

▶ **Lemma 16.** $\max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))} \mathrm{EIC}_{P_{\mathrm{TO}}(\tau)}(C, \mathcal{S}) = O(\tau n \log n)$.

Thus, we obtain the main theorem.

▶ **Theorem 17.** *For any $\tau \in \mathbb{N}^+$, $P_{\mathrm{TO}}(\tau)$ is an $(O(\tau n \log n), \Omega(n^\tau))$-LS-LE protocol.*

6 Conclusion

We gave a time-optimal LS-LE protocol in the population protocol model. Given a design parameter $\tau \ge 1$ and integer $N \ge n$ such that N is at most polynomial in n, the proposed protocol elects the unique leader within $O(\tau \log n)$ parallel time starting from any configuration and keeps it for $\Omega(n^\tau)$ parallel time, both in expectation.

References

1 Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. Time-space trade-offs in population protocols. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2560–2579. SIAM, 2017.

2 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2221–2239. SIAM, 2018.

3 Dan Alistarh, Bartłomiej Dudek, Adrian Kosowski, David Soloveichik, and Przemysław Uznański. Robust detection in leak-prone population protocols. In *International Conference on DNA-Based Computers*, pages 155–171. Springer, 2017.

4 Dan Alistarh and Rati Gelashvili. Polylogarithmic-time leader election in population protocols. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming*, pages 479–491, 2015.

5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

6 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

7 Dana Angluin, James. Aspnes, Michael J Fischer, and Hong Jiang. Self-stabilizing population protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):13, 2008.

8 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proceedings of 52nd Annual ACM Symposium on Theory of Computing*, 2020.

9 Janna Burman, Ho-Lin Chen, Hsueh-Ping Chen, David Doty, Thomas Nowak, Eric Severson, and Chuan Xu. Time-optimal self-stabilizing leader election in population protocols. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 33–44, 2021.

10 Shukai Cai, Taisuke Izumi, and Koichi Wada. How to prove impossibility under global fairness: On space complexity of self-stabilizing leader election on a population protocol model. *Theory of Computing Systems*, 50(3):433–445, 2012.

11 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 53–59, 2019.

12 Hsueh-Ping Chen and Ho-Lin Chen. Self-stabilizing leader election in regular graphs. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 210–217, 2020.

13 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.

14 Leszek Gąsieniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2653–2667. SIAM, 2018.

15 Leszek Gąsieniec, Grzegorz Stachowiak, and Przemyslaw Uznanski. Almost logarithmic-time space optimal leader election in population protocols. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 93–102. ACM, 2019.

16 Taisuke Izumi. On space and time complexity of loosely-stabilizing leader election. In *International Colloquium on Structural Information and Communication Complexity*, pages 299–312, 2015.

17 Yuichi Sudo and Toshimitsu Masuzawa. Leader election requires logarithmic time in population protocols. *Parallel Processing Letters*, 30(01):2050005, 2020.

18 Yuichi Sudo, Junya Nakamura, Yukiko Yamauchi, Fukuhito Ooshita, Hirotsugu. Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election in a population protocol model. *Theoretical Computer Science*, 444:100–112, 2012.

19 Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Time-optimal leader election in population protocols. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

20 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Loosely-stabilizing leader election on arbitrary graphs in population protocols without identifiers nor random numbers. In *International Conference on Principles of Distributed Systems*, 2015.

21 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. Loosely-stabilizing leader election for arbitrary graphs in population protocol model. *IEEE Transactions on Parallel and Distributed Systems*, 30(6):1359–1373, 2018.

22 Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, Toshimitsu Masuzawa, Ajoy K Datta, and Lawrence L Larmore. Loosely-stabilizing leader election with polylogarithmic convergence time. *Theoretical Computer Science*, 806:617–631, 2020.

23 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Self-stabilizing population protocols with global knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):3011–3023, 2021.

24 Daisuke Yokota, Yuichi Sudo, and Toshimitsu Masuzawa. Time-optimal self-stabilizing leader election on rings in population protocols. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, (to appear), 2021.

A Omitted proofs

We prove Lemmas 15 and 16 in Sections A.1 and A.2, respectively.

A.1 Holding Time

We prove Lemma 15, which claims $\min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{TO}}(\tau)}(C) = \Omega(n^{\tau+1})$.

Proof of Lemma 15. Let C_0 be any configuration in $\mathcal{S}$ and let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. Since the unique leader v_l in C_0 satisfies $v_l.\mathtt{detect} = 1$ and r_{mid} is a sufficiently large $\Theta(\tau \log n)$ value, by Lemma 2 with $k = \tau$ and the third claim of Lemma 7, all agents detect the existence of a leader quickly with very high probability. Therefore, by the first and second claims of Lemma 7, it holds with very high probability that Ξ quickly reaches a configuration $C' \in \mathcal{C}_{EL\geq}$ where v_l is the unique leader and no agent has a higher level than $v_l.\mathtt{level}$. This is because v_l goes to the election phase exactly once before Ξ reaches C' with very high probability, and the level of every agent is initialized to zero when it goes to the election phase. Thereafter, v_l never becomes a follower before it goes back to the check phase and goes to the election phase again; this is because only a leader can increase $\max_{v \in V} v.\mathtt{level}$. By Lemma 8, Ξ quickly enters $\mathcal{A}_{CH\geq}$ again and v_l does not move to the election phase from the check phase during this period with very high probability. At this time, from the above discussion, v_l is still the unique leader in the population and $v_l.\mathtt{detect} = 1$ holds. This means that the population has come back to $\mathcal{S}$.

Now, we observed that an execution of P_{TO} under the uniformly random scheduler $\mathbf{\Gamma}$ starting from any configuration in $\mathcal{S}$ goes back to a configuration in $\mathcal{S}$ after $\Theta(\tau n \log n)$ steps and v_l is always the unique leader during this period with very high probability. Therefore, letting $X = \min_{C \in \mathcal{S}} \mathrm{EIH}_{P_{\mathrm{TO}}(\tau)}(C)$, we have $X \geq (1 - O(n^{-\tau}))(\Theta(\tau n \log n) + X)$. Solving this inequality gives $X = \Omega(\tau n^{\tau+1} \log n) = \Omega(n^{\tau+1})$. ◀

A.2 Convergence Time

We prove Lemma 16, which claims $\max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))} \mathrm{EIC}_{P_{\mathrm{TO}}(\tau)}(C, \mathcal{S}) = O(\tau n \log n)$.

Proof of Lemma 16. Let C_0 be any configuration in $\mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))$ and let $\Xi = \Xi_{P_{\mathrm{TO}}(\tau)}(C_0, \mathbf{\Gamma})$. It suffices to show that Ξ enters $\mathcal{S}$ quickly with a constant probability; this is because, letting $Y = \max_{C \in \mathcal{C}_{\mathrm{all}}(P_{\mathrm{TO}}(\tau))} \mathrm{EIC}_{P_{\mathrm{TO}}(\tau)}(C, \mathcal{S})$, it yields $Y \leq O(\tau n \log n) + (1 - \Omega(1))Y$, and this inequality gives $Y = O(\tau n \log n)$.

We can assume $C_0 \in \mathcal{C}_{EL\geq}$ by Lemmas 6 and 7. By Lemma 8, Ξ reaches a configuration $C' \in \mathcal{A}_{CH\geq}$ within $O(\tau n \log n)$ steps and no agent goes to the election phase from the check phase during this period with very high probability. An agent executes $\mathtt{detect} \leftarrow \mathtt{leader}$ when it goes back to the check phase (See Table 2). Therefore, after Ξ reaches C', at least one follower becomes a leader when it goes to the election phase if there exists no leader in C'. Moreover, the agents initialize their $\mathtt{level}$ and $\mathtt{done}$ to 0 when they move to the election phase. Therefore, by Lemmas 5, 7, 8, and 9, Ξ quickly reaches a configuration $C'' \in \mathcal{C}_{EL\geq}$ where exactly one leader, say v_l, has the maximum level with a constant probability. Thereafter, by Lemmas 8 and 13, Ξ reaches a configuration in $\mathcal{A}_{EL}$ where only v_l is a leader within $O(n \log n)$ steps with probability $1 - o(1)$. In the next $O(\tau n \log n)$ steps, Ξ enters $\mathcal{S}$ with very high probability by Lemma 8. Thus, we conclude that Ξ enters $\mathcal{S}$ within $O(\tau n \log n)$ steps with a constant probability. ◀

Efficient Distribution of Quantum Circuits

Ranjani G Sundaram
Department of Computer Science, Stony Brook University, New York, NY, USA

Himanshu Gupta
Department of Computer Science, Stony Brook University, New York, NY, USA

C. R. Ramakrishnan
Department of Computer Science, Stony Brook University, New York, NY, USA

Abstract

Quantum computing hardware is improving in robustness, but individual computers still have small number of qubits (for storing quantum information). Computations needing a large number of qubits can only be performed by distributing them over a network of smaller quantum computers. In this paper, we consider the problem of distributing a quantum computation, represented as a quantum circuit, over a homogeneous network of quantum computers, minimizing the number of communication operations needed to complete every step of the computation. We propose a two-step solution: dividing the given circuit's qubits among the computers in the network, and scheduling communication operations, called migrations, to share quantum information among the computers to ensure that every operation can be performed locally. While the first step is an intractable problem, we present a polynomial-time solution for the second step in a special setting, and a $O(\log n)$-approximate solution in the general setting. We provide empirical results which show that our two-step solution outperforms existing heuristic for this problem by a significant margin (up to 90%, in some cases).

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies; Computing methodologies → Distributed algorithms

Keywords and phrases Distributed Quantum Computing, Hypergraph Min-Cut

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.41

Funding This work was partially supported by NSF CNS Award # 1815306.

1 Introduction

Motivation and Driving Problem. Over the recent past, there has been a steady increase in the capacity of quantum computing hardware. A crucial obstacle to achieving the full potential of quantum computing is the limited number of *qubits*, which are basic stores of quantum information, in any single quantum computer (QC). Distributing a large quantum computation over a network of QCs is a way to overcome this obstacle [7, 5]. Distributing a computation among a network of QCs will be important to realize the promise and power of quantum computation. Quantum *circuits* form a useful abstraction between higher-level quantum *programs* written in languages such as Qiskit [14] and Quipper [15], and lower-level computing hardware. In this paper, we consider the problem of *optimally distributing a given quantum circuit for evaluation over a network of QCs.* This distribution involves mapping the qubits in the circuit to individual QCs such that the communication needed to perform operations whose operands span different QCs is minimized. The distribution problem is novel to quantum computing in two important ways:

1. Quantum circuits have an explicit linear structure that makes it possible to determine the cost of a distribution before the circuit is evaluated. In contrast, classical programs have conditionals and loops where data dependencies cannot be predicted before execution.
2. *Entanglement*, which is a phenomenon unique to quantum computing, enables modes of communication that permit efficient sharing of information between operations.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 41; pp. 41:1–41:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The above differences enable novel solutions to the problem of distributing quantum computation (see §2 for an elaboration of these key concepts).

Efficient techniques for executing a single operation with operands that span different QCs in a network have been studied before [11, 18, 19]. More recently, [8] posed the problem of finding an optimal distribution of a quantum circuit under a naive communication model in terms of balanced graph partitioning, and solved it using well-known heuristics for that problem [12]. A formulation with a refined communication model that exploits quantum entanglement was considered in [3], where the optimization problem was shown to be intractable and solved using hypergraph partitioning heuristics [1]. A more detailed comparison with related work is in §2.3.

Our Approach and Contributions. We pose the problem of optimal distribution of quantum circuit evaluation in two steps (see overview in §3). In the first step, we partition the qubits of the circuit using ordinary balanced graph partitioning, but with the edge weights determined by accurately estimating the cost of placing the corresponding qubits in different partitions. In the second step, we solve the problem of optimal placement of operations for a given partition. The main contributions of the paper are:

- For an appropriately formulated DQC problem of optimizing distributed evaluation of quantum circuits, we develop an efficient heuristic that outperforms the prior work by a significant margin (up to 90%) across a wide range of parameters (see §6).
- Our algorithm for the DQC problem is a two-step heuristic. For the second step of covering gates using minimum number of communication primitives, we design an optimal algorithm for a specialized setting (see §4), and an $O(\log n)$-approximation algorithm for the general setting (see §5).

We defer the proofs of all lemmas and theorems to Appendix A.

2 Background and Related Works

2.1 Quantum Computation and Communication

This paper's technical development can be followed with a high-level understanding of three of quantum computing/communication concepts: the structure of quantum circuits, properties of certain quantum gates, and characteristics of quantum communication that are directly used in our development. In the following, we give an overview of these three concepts. For a more thorough review of this area, the reader is referred to standard sources such as [13].

Quantum Circuits. Quantum computation is typically abstracted as a *circuit*, where horizontal "wires" represent *qubits* which carry quantum data, and operations on the qubits performed by vertical "gates" connecting the operand wires. Quantum computers (QCs) evaluate a circuit by applying the gates in the left-to-right order, so this circuit can also be understood as a sequence of machine-level instructions (gates) over fixed number of data cells (qubits).

Analogous to classical Boolean circuits, there are several universal gate sets for quantum computation: any quantum computation can be expressed by a circuit consisting only of gates from a universal gate set. In particular, a special binary gate called *CZ* along with the set of all possible unary gates forms a universal gate set; we make use of this universal gate set in this paper. Fig. 1 shows the pictorial representation of an example circuit, consisting only of unary gates (boxes) and *CZ* gates (vertical connectors). Quantum circuits also comprise

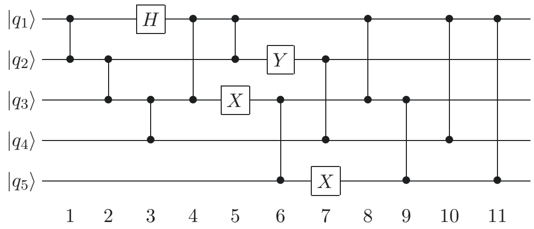


Figure 1 Running Example Quantum Circuit.

of measurement operations which yield classical bits as results, and conditional operations that are guarded by classical bit values. For the purposes of this paper, these operations can be treated as unary operations.

Quantum Communication. If a given quantum circuit is to be evaluated in a distributed fashion over a network of QCs, we have to first distribute the qubits over the QCs. But such a distribution may induce gates in the circuit to span different QCs. To execute such *non-local* gates, we need to bring all operands' values into a single QC via quantum communication. However, physical transmission of arbitrary qubits can incur *irreparable* communication errors, as the *No Cloning Theorem* [17, 9] proscribes making *independent* copies of arbitrary qubits. An alternative approach to communicate qubits is via *teleportation* [4], which requires an a priori distribution of maximally-entangled pair (MEP) of qubits (e.g., Bell Pair) over the two nodes. With an MEP distributed over nodes A and B, teleportation of a qubit state from A to B can be accomplished using classical communication and local gate operations, while consuming/destroying the MEP.

Creating "Linked Copies" of a Qubit via Cat-Entanglement. Another means of communicating qubit states is by creating *linked copies* of a qubit across QCs, via *cat-entanglement* operations [11, 19] which, like teleportation, require a Bell Pair to be shared *a priori.* These linked copies are particularly useful in efficient distributed evaluation of circuits involving only CZ and unary gates, as follows. *CZ* gates have two special properties: (i) we can freely replace an operand q with a linked copy of q; and (ii) *CZ* operations *commute* with cat-entanglement operation. As a consequence, the linked copies act like shared memories that continue to remain "in sync" across QCs as long as only *CZ* operations are executed on them. Unlike *CZ* gates, unary gates destroy the cat-entanglements in general.[1] Before applying a unary operation on q, we have to disentangle any linked copies via a dual operation called *cat-disentanglement* which doesn't require a Bell Pair.

2.2 Our Model of Distributed Evaluation of Quantum Circuits

Our model for distributed evaluation of quantum circuits is as follows.

1. Without loss of generality, we assume, that the given (centralized) circuit is composed only of unary and CZ gates. Circuits using other gate sets can be rewritten to this form, with only a polynomial expansion in size.
2. We assume that memory for Bell pairs used in cat-entanglement (called ***ebits***) are distinct from that used for computation qubits, as in prior works [3].

[1] An important exception are phase-shift gates which also preserve cat-entanglement.

3. Our network model consists of homogeneous set of QCs: the memory capacity for computation/data qubits is nearly the same in all QCs. Thus, when distributing an input circuit, we partition the number of qubits almost-uniformly across the given QCs.
4. We consider only a "static" assignment of qubits to QCs, i.e., each qubit has a home where all the unary operations are performed.
5. Due to the advantages listed in §2.1, we use cat-entanglement as the only mechanism to communicate qubits.
6. Gates in the given quantum circuit are executed in the order they appear in the circuit: no further optimizations such as gate reordering are done during distributed evaluation.

Cost Model and Objective. Let us assume that the given (centralized) quantum circuit is already optimized in the sense that successive gates on different operands are executed in parallel. For such an optimized circuit, if we disallow any gate reordering optimization (#6 above), it is easy to see that all distributed evaluation schemes would incur the same computational cost/time. Thus, our optimization objective in distribution of a given quantum circuit is to minimize the communication cost, i.e., the number of cat-entanglement operations needed to evaluate the gates. As mentioned above, cat-entanglements require a priori shared MEPs, which is an expensive resource – generating them can incur substantial latency due to the stochastic nature of underlying processes [10, 16]. Note that cat-disentanglement operations only require local operations and classical communication, and furthermore, only done following prior cat-entanglements. Hence, their cost can be ignored (or folded into the cost of cat-entanglements).

2.3 Related Work

The problem of implementing non-local gates in distributed quantum circuits was considered in [11]. That paper showed necessary and sufficient communications needed to implement several gates non-locally. In particular, they used the idea of cat-entanglement to share linked copies across QCs. Such sharing was further explored in [19], which also introduced the terms "cat-entanglement" and "cat-disentanglement". The paper generalized sharing to the multi-partite case, by essentially composing cat-entanglement operations. Both [11] and [19] focus on optimal implementation of given non-local gates.

In contrast, [8] and [3] focus on optimizing the assignment of qubits to QCs in order to minimize communication costs. A teleportation-based model is used in [8]: non-local operands of a gate are teleported to a common QC for the gate operation, and then teleported back to their original locations. The paper poses the optimization problem in terms of balanced k-partition of a weighted graph with qubits as vertices. The paper uses the graph partitioning algorithm of [12] to derive good partitions.

The closest work to ours is that of [3], where cat-entanglement operations are used to share linked copies of qubits. The optimal assignment problem is posed in terms of balanced *hypergraph* partitioning, with hyperedges capturing the set of gates that can be executed with linked copies. The assignment problem is shown to be intractable via reduction from the hypergraph balanced k-min-cut problem. In contrast, we have developed a two-step heuristic, which uses a simpler min-cut problem in the first step and a coverage problem in the second step that can be solved optimally or near-optimally. We compare our work empirically with theirs in §6.

3 DQC Problem Formulation and High-Level Algorithm

In this section, we formally define the problem of efficient distribution of quantum circuits, and give a high-level description of our proposed algorithm.

Quantum Circuits. We use $q_1, q_2, \ldots, q_n$ to denote *qubits*. Quantum circuits are composed of gates. Based on §2, we consider the universal gate set with (binary) *CZ* and unary gates. Since we only use one type of binary gate and the *type* of unary gate does not play a role in our context, we need to only represent the operands of each gate and not the gate itself. We use the below notion of an *abstract* quantum circuit that elides the gate information.

▶ **Definition 3.1** ((Abstract) Quantum Circuit)**.** *Given a set of qubits $Q = \{q_1, q_2, \ldots\}$, an* abstract quantum circuit *C over Q is a sequence of gates $\langle g_1, g_2, \ldots \rangle$ where each g_k is one of:*

- *(q_i, q_j): the pair of operands for a CZ gate that occurs as the k-th gate.*
- *q_i: the operand of an unary gate that occurs as the k-th gate.*

Throughout the article, we thus represent binary gates in a circuit as triplets (q_i, q_j, t), and unary gates as pairs (q_i, t), where t is the index/timestamp of the gate in the circuit.

3.1 Distributed Quantum Circuit (DQC) Problem

Our goal is to determine an efficient distribution of a given quantum circuit, over a given network of QCs. Efficient distribution essentially entails two tasks: distributing the qubits over the distributed QCs, and then executing the gates efficiently over the distributed QCs. We represent the distribution of qubits over the distributed QCs as a *partition* of the qubits. We also define a concept of *migrations* which represent the cat-entanglement operations; each migration may facilitate execution of one or more gates, which is captured via a notion of coverage of gates by migrations.

Informally, the DQC problem is to (i) partition the given circuit across distributed QCs, and (ii) for the given partition, determine a small set of migrations that are sufficient to execute/cover all the non-local gates in the circuit. The overall goal of the DQC problem is to minimize the number of cat-entanglements (or migrations, as we represent them as). Below, we formalize the notions of partitions, migrations, and execution of gates by migrations (via a notion of coverage), before defining the DQC problem formally.

Partitioning. Distributing a quantum circuit first entails assigning qubits to quantum computers in the network. In this paper, we implicitly assume homogeneous quantum computers, and thus, we consider only near-balanced partitions of the qubits across QCs.

▶ **Definition 3.2** (Partition)**.** *Given integer $k > 0$ and real $\nu > 1$, a balanced (k, ν)-partition of a finite set S divides S into k components i.e., disjoint subsets of S, such that each component is of size at most $\nu \cdot \frac{|S|}{k}$.*

A partition can be represented by a total function π that maps S to a set of labels $P = \{p_1, p_2, \ldots, p_k\}$.

For our DQC problem, we consider (k, ν)-balanced partitions of the set of qubits Q using the set of QCs, $P = \{p_1, p_2, \ldots, p_k\}$, as the partition labels. Note that such a partition assigns qubits to QCs. Throughout this paper, we refer to quantum computers p_i as partitions, and $\pi(q)$ as the *home-partition* of q. In addition, we refer to a quantum circuit with an already given partition as a *partitioned circuit*.

Migrations. A migration essentially represents the cat-entanglement operation to create a *linked* copy of a qubit in a partition other than its home-partition. We allow migrations of a qubit q_i to occur only initially (i.e., $t = 0$) or right after a unary operation on q_i. This is without loss of generality, because, since cat-entanglement commutes with CZ, we can move any migration to the most recent unary operation or $t = 0$. Also, disentanglement operations are implicit – done, if needed, immediately before any unary operations, so are not represented.

▶ **Definition 3.3** (Migration). *Given a set of qubits Q, abstracted circuit C and partition P, a* migration *is a triple (q_i, p_k, j) where $q_i \in Q$, $p_k \in P$ such that $p_k \neq \pi(q_i)$ (i.e. not the home-partition of q_i), and either $j = 0$ or $(q_i, j) \in C$, i.e., j is the index of an unary gate on q_i in C.*

Coverage of Gate by Migration(s). Consider a binary gate (q_i, q_j, t), where $\pi(q_i) \neq \pi(q_j)$ i.e., the operands are in different partitions. We refer to such gates as *non-local* gates. To execute such a non-local binary gate, we can: (i) Migrate q_i to the home partition of q_j, (ii) Migrate q_j to the home partition of q_i, or (iii) Migrate both q_i and q_j to a third partition where the gate operation can then be performed. In each of the above cases, since unary gates do not commute with migrations, we need to ensure that there are no unary gate operations on the migrated qubit between the migration and the binary gate operation being covered. This notion is formalized below, in terms of coverage of a gate via migrations.

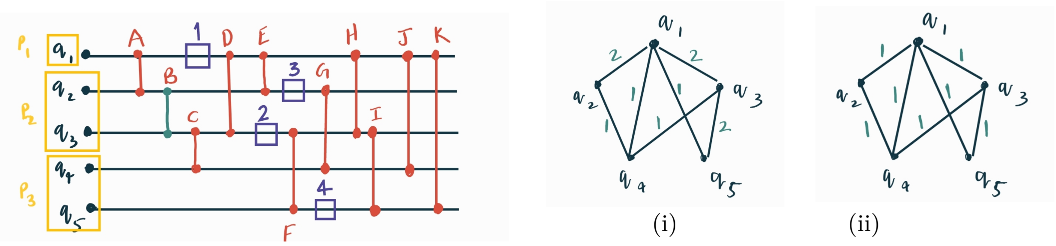
▶ **Definition 3.4** (Coverage). *For a given partition π, a binary gate (q_i, q_j, t) can be covered by a single or a pair of migrations in one of the following ways.*

1. *By $(q_i, \pi(q_j), t')$, if $t' < t$ and there are no unary gates on q_i between t' and t; or*
2. *By $(q_j, \pi(q_i), t')$, if $t' < t$ and there are no unary gates on q_j between t' and t; or*
3. *By the pair $\{(q_i, p_k, t'), (q_j, p_k, t'')\}$, if $t', t'' < t$ and there are no unary gates on q_i between t' and t and on q_j between t'' and t.*

Note that the third condition above allows a gate (q_i, q_j, t) to be executed at a partition other than the home-partitions of either of the operand qubits (i.e., by migrating *both* the operand qubits to a third partition.

DQC Problem Formulation. Given a quantum circuit, the number k of distributed QCs, and a partitioning factor $\nu \geq 1$, the DQC problem is to (k, ν)-partition the qubits into the distributed QCs and determine the set of migrations that cover the non-local gates of the partitioned circuit, such that the total number of migrations required is minimized. The DQC problem is known to be NP-hard by a reduction from the hypergraph min-cut problem [3].

DQC Example. Consider the running example of our quantum circuit from Fig. 1. Fig. 2a shows the distribution of qubits into three partitions: $p_1 = \{q_1\}$, $p_2 = \{q_2, q_3\}$ and $p_3 = \{q_4, q_5\}$. The non-local binary gates are represented by red vertical lines, and the local ones by green lines. Let $t_i (i \geq 1)$ be the timestamp associated with the i-th unary gate. The set of all possible migrations for this partitioned circuit is: $\{(q_1, p_2, 0), (q_1, p_3, 0), (q_2, p_1, 0), (q_2, p_3, 0), (q_3, p_1, 0), (q_3, p_3, 0), (q_4, p_1, 0), (q_4, p_2, 0), (q_5, p_1, 0), (q_5, p_2, 0), (q_1, p_2, t_1), (q_1, p_3, t_1), (q_3, p_1, t_2), (q_3, p_3, t_2), (q_2, p_1, t_3), (q_2, p_3, t_3), (q_5, p_1, t_4), (q_5, p_2, t_4)\}$. A example of a set of migrations that can cover all the non-local gates is: $\{(q_1, p_2, 0), (q_4, p_2, 0), (q_5, p_2, 0), (q_1, p_2, t_1), (q_5, p_2, t_4)\}$. This set of migrations execute all the non-local gates in partition p_2.



(a) Partitioning of our running example quantum circuit; the three partitions are shown by yellow boxes.

(b) Edge weights on the graph used for partitioning the running example. (i) Simple weights based on the total number of binary gates between the qubits, and (ii) Refined edge weights based on the optimal `MS-HC` algorithm in §4.

Figure 2 Partitioning and Edge Weights, on the running example.

3.2 Two-Step DQC Algorithm

In a recent work, Martinez and Heunen [3] reduce the `DQC` problem to a balanced k-min-cut of an appropriate hypergraph, and thus, use a hypergraph-partitioning heuristic to solve the problem. Here, we propose a use natural two-step procedure – in effect, reducing the `DQC` problem to a sequence of two simpler problems; in fact, we show that the second step can be solved near-optimally and even optimally in a special setting. In particular, our proposed algorithm consists of the following two steps: (i) First, we compute a "good" (k, ν)-partition of the qubits for distribution over the k QCs; (ii) Then, we determine a minimal set of migrations required to cover all the non-local gates of the partitioned circuit.

Step 1: Partitioning Qubits into QCs/Partitions. To compute a partition that will intuitively yield a small set of migrations in the second step, we compute the partitioning of the qubits by solving a balanced k-min-cut problem over an edge-weighted graph over qubits as vertices. In particular, given a circuit C, we define a weighted graph $G = (V, E)$ where V is the set of qubits in C and $E = \{(q_i, q_j) | (q_i, q_j, t) \in C\}$. The weight on each edge intuitively represents the cost of keeping the qubits in different partitions; at its simplest, the weight of an edge (q_i, q_j) can be the number of binary gates of the type $(q_i, q_j, \_)$ in C for some t. A more refined weight function is defined at the end of this section.

In essence, in the first step, we partition the qubits by computing a balanced min-k-cut of the of the above weighted graph. The balanced min-k-cut problem is NP-hard even for $k = 2$, with no known approximation algorithms. Therefore, we consider an approximate version of the problem, viz., (k, ν)-balanced graph partitioning for $\nu > 1$ as defined in Def. 3.2. There are several works that address the (k, ν)-balanced min-cut problem. Notably, [2] gives a $O(\log^2 n)$-approximation algorithm for $\nu = 1 + \epsilon$ (ϵ is arbitrarily small) when k is a constant. In our evaluation, we use a third-party graph partitioning algorithm called KaHyPar [1].

Step 2: Selecting Migrations. Given a partitioned circuit, we then find the smallest set of migrations that cover all the non-local gates. This minimization problem can be solved optimally in polynomial time when restrict each gate to be executed only in the home-partition of one of its operands (§ 4). For the general case, we provide an $O(\log N)$-approximation algorithm (where N is the number of binary/CZ gates in the circuit C).

Refined Weight Function in Step 1. The total number of CZ gates between two qubits q_i and q_j, originally proposed as the weights of edges in Step 1 above, is actually just an upper bound on the number of migrations needed to cover those gates. A more accurate cost will be the actual/optimal number of migrations needed to cover $(q_i, q_j, \_)$ gates if q_i and q_j were in separate partitions. We use the above intuition to construct an *induced* circuit $C'_{(i,j)}$ consisting only of gates on q_i and q_j. We distribute $C'_{(i,j)}$ such that the two qubits q_i and q_j are in different partitions and compute the *minimum* number of migrations needed to cover all the binary gates in $C'_{(i,j)}$. The minimum number of migrations in $C'_{(i,j)}$ can be computed using the optimal MS-HC algorithm described in §4, and we use this minimum value as the weight on the edge $(q_i,\, q_j)$ of the weighted graph used in Step 1. See Fig. 2b.

4 Optimal Selection of Migrations under Home-Coverage

In this section, given a quantum circuit and a partitioning of its qubits to computers/partitions, we design an efficient algorithm to compute the optimal set of migrations needed to execute the partitioned circuit. The algorithm developed in this section is used as a subroutine in the Two-Step DQC Algorithm presented in §3.

Here, we will assume that a gate can only be executed at the home-partition of one of its operand-qubits; this restriction helps minimize execution memory needed at individual partitions, and in addition, simplifies the migration-selection problem sufficiently that we can design an optimal algorithm for the problem of selection of migrations to cover all gates. We will relax this assumption in the next section where we describe an algorithm for the migration-selection problem in the more general setting.

We start with formally incorporating the above assumption in our problem formulation by restricting the original definition (Def. 3.4) of coverage of gates by migrations.

Home Coverage of Gates by Migrations. Given a quantum circuit, recall the definition (Def. 3.4) of coverage of gates by migrations, for a given partition π. Therein, we defined that a gate can be covered by migration(s) in many ways. To impose the condition that each gate be executed only at a home-partition of its operand qubits, we restrict the definition of coverage by allowing a migration to cover a gate only if it migrates one of the operand qubits to the home-partition of the other qubits. We define the notion of home-coverage below.

▶ **Definition 4.1** (Home-Coverage). *For a given partition π, a binary gate gate (q_i, q_j, t) can be home-covered by a migration in one of the following ways.*

1. *By $(q_i, \pi(q_j), t')$, if $t' < t$ and there are no unary gates on q_i between t' and t; or*
2. *By $(q_j, \pi(q_i), t')$, if $t' < t$ and there are no unary gates on q_j between t' and t.*

Memory Conservation by Home-Coverage. Restricting execution of gates at home partitions of its operands is one way to implicitly minimize/constrain the size of execution memory needed. For example, consider a circuit with n qubits $\{q_1, q_2, \ldots, q_m\}$, no unary gates, and $n-1$ binary CZ-gates of the type (q_i, q_{i+1}, t_i) for some t_i's. Suppose that each qubit is distributed to a different partition. In this case, a potential set of migrations to cover the gates could be to migrate all qubits to a *single* partition and execute all the gates there. However, this would require $n-1$ units of additional memory to hold all the migrated qubits. In contrast, the alternate solution restricted by home-coverage would migrate qubit q_i to $\pi(q_{i+1})$ for each i, requiring only one additional memory at each partition. In addition, as mentioned above, restricting the coverage of gates to home-coverage allows design of an optimal algorithm for selection of migrations, and thus, an efficient overall solution for the DQC problem as observed in our evaluations.

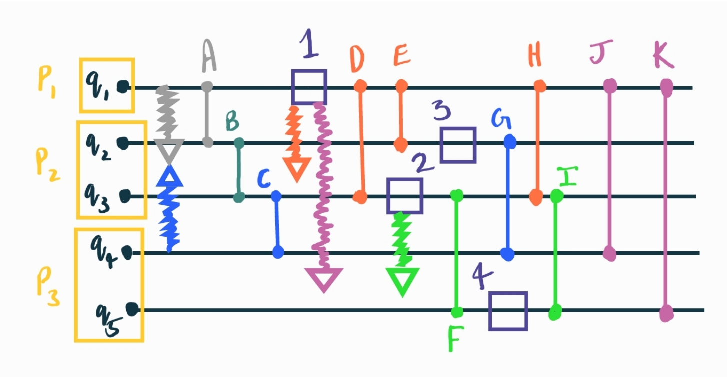


Figure 3 Illustrating the MS-HC Problem. The set of gates (straight vertical lines) covered by a migration (squiggly lines) m are shown in the same color as m. E.g., gates D, E and H are covered by the migration (q_1, p_2, t_1).

MS-HC Problem. Migration Selection under Home-Coverage. Given a quantum circuit C and a partitioning of C's qubits to given partitions/QCs, the MS-HC problem is to determine the smallest set of migrations that can execute the gates of the partitioned circuit in the home partitions of one of its operands. More formally, let Q be the set of qubits in a given circuit C, P be the set of given partitions, π be the partitioning function, and M be the set of all potential migrations (as per Def. 3.3). The MS-HC problem is to select a smallest set of migrations M such that each gate in C is home-covered by some migration in M.

We start with illustrating the MS-HC problem via an example. Then, we design an optimal algorithm for the MS-HC problem. Recall our running example of a quantum circuit from Example 2a. As shown in Fig. 3, one solution to the MS-HC problem for the given partition of the qubits is $\{(q_1, p_2, 0), (q_4, p_2, 0), (q_1, p_2, t_1), (q_1, p_3, t_1), (q_3, p_3, t_2)\}$, where as before t_i is the timestamp of the i-th unary gate. This solution can be easily verified as optimal.

4.1 Optimal MDS-HC Algorithm

We start with a high-level description and intuition of the proposed optimal algorithm, followed by the proofs of two key claims needed to design the algorithm.

High-level. The MS-HC problem can be easily formulated as the well-known set-cover problem. However, MS-HC is in fact a very special case of the set-cover, allowing us to design a polynomial-time optimal algorithm. In particular, first, we show that each gate (element) is home-covered by exactly two migrations (sets), and thus the MS-HC problem can be formulated as a vertex-cover problem in an appropriate graph G_{HC}, where migrations are represented as nodes and gates as edges. Second, we show that the graph G_{HC} is actually a bipartite graph, by showing that it has no odd length cycles. Thus, the MS-HC problem reduces to the vertex-cover problem in the bipartite graph G_{HC}, and thus, solvable optimally in polynomial time.

Each Gate is Home-Covered by Two Migrations. Recall from Def. 3.3 that a migration is a triplet (q, P, t), where the qubit q is being migrated to partition P at time t, and we only allow migrations with a timestamp t of either 0 or corresponding to some unary gate on qubit q. The below lemma shows that each gate is home-covered by exactly two such migrations.

▶ **Lemma 4.2.** *For a given partitioned quantum circuit, a binary gate in the circuit is home-covered by exactly two migrations.*

It is easy to see that the above lemma holds for the partitioned circuit of our running example. See Figure 5 in Appendix A.

Bipartite Graph over Migrations. Based on the intuition from Fig. 5, we can construct a graph $G_{HC}(V = M, E = T)$ where T is the set of all non-local gates (i.e., gates with different home-partitions of the operands) in the given partitioned circuit and M is the set of all migrations as per Def. 3.3 (i.e., with a timestamp of 0 or a unary gate on the migrated qubit). More formally, an edge/migration (q_i, q_j, t) connects the unique vertices $(q_i, \pi(q_j), t_i)$ and $(q_j, \pi(q_i), t_j)$ where t_i and t_j are as defined in Lemma 4.2. Based on this graph representation of home-coverage of gates by migrations, we can solve the `MS-HC` problem by computing the optimal vertex-cover of edges in G_{HC}, since a vertex cover of G_{HC} corresponds exactly to a set of migrations that home-cover all the given gates. Below, we prove that, the G_{HC} graph is actually a bipartite graph – which allows us to compute the optimal vertex-cover of G_{HC} in polynomial time.

▶ **Lemma 4.3.** *For a given partitioned circuit, we claim that the G_{HC} graph, as defined above, has no odd-length cycles, and is thus a bipartite graph.*

Optimal Algorithm for MS-HC. Based on the above two lemmas, we can now solve the `MS-HC` problem by computing an optimal vertex cover of the bipartite graph G_{HC} for a given partitioned circuit. The pseudo-code of the overall algorithm is given in Appendix A.4.

▶ **Theorem 4.4.** *Given a partitioned quantum circuit, Algorithm 2 returns an optimal set of migrations that home-cover all the non-local gates of the given partitioned circuit.*

5 Near-Optimal Selection of Migrations under General Coverage

In this section, we relax the assumption made in the previous section, i.e., allow execution of non-local gates of a partitioned circuit in partitions different from any of the home-partitions of its operands. We do this by using the original notion of coverage defined in Def. 3.4, where a non-local gate may also be executed by migrating both of its operand qubits to a third partition. In such a general setting, for a given partitioned circuit, we consider the problem of selecting a minimum number of migrations to cover all non-local gates, and present a polynomial-time approximation algorithm.

MS-GC Problem: Selection of Migrations to Cover Gates. Given a partitioned quantum circuit, the `MS-GC` problem is to determine a set a migrations of minimum size that covers all the non-local binary gates of the given partitioned circuit.

The above `MS-GC` problem generalizes the `set-cover` problem in some sense, as it allows an element to be covered by a pair of sets (i.e., an element is covered only if both the sets are selected). However, the `MS-GC` problem also has a special structure, which makes proving its intractability non-trivial; however, we conjecture the `MS-GC` problem to be NP-hard. In either case, since the objective function is not submodular, the simple greedy algorithm that iteratively selects the migration with most "benefit" does not offer a performance guarantee. Below, we will design a polynomial-time approximation algorithm for the `MS-GC` problem. We start with a definition.

▶ **Definition 5.1** (Same-Partition Set (of Migrations)). *A set of migrations M is called a* same-partition set *if for every pair of migrations (q_1, p_1, t_1) and (q_2, p_2, t_2) in M, we have $p_1 = p_2$.*

5.1 Algorithm G*: Approximation Algorithm for the MS-GC Problem

Our proposed algorithm G^* is a greedy algorithm that picks, at each iteration, a same-partition set of migrations with highest benefit-density (as defined below), until all the non-local gates of the given partitioned circuit are covered by the picked set of migrations. Note that, at each stage, the G^* algorithm may pick more than one migration. We will later develop a procedure (Algorithm 1) to select a same-partition set with highest benefit-density, which form a single iteration of the G^* algorithm. Below, we first show that G^* will deliver an $O(\log N)$-approximate solution to the MS-GC problem, where N is the total number of non-local gates in the given partitioned circuit. We now define the notion of benefit and benefit-density of a set of migrations.

▶ **Definition 5.2** (Benefit; Benefit-Density). *Consider a stage/iteration in the G^* algorithm, where a set of migrations have already been selected. For a set X of migrations, we define its* benefit *$B(X)$ as the number of (non-local) gates covered by X that have not been covered yet by the set of migrations already selected in previous iterations. We define the* benefit-density *of X as $B(X)/|X|$).*

▶ **Theorem 5.3.** *For the MS-GC problem, the G^* algorithm delivers a solution with at most $|O| \ln N$ number of migrations, where O is the optimal solution and N is the total number of non-local gates in the given partitioned circuit.*

5.2 Dividing a Set into Same-Partition Subsets

We now show that a set of migrations can be divided into a disjoint collection of same-partition subsets such that the benefit of the original set is at most the summation of the benefit of the subsets. We start with a formal definition and on observation.

▶ **Definition 5.4** (Benefit Graph). *The benefit-graph, denoted by $B_M(V, E)$, for a set of migrations M at a certain stage of G^* is defined as follows. The set of vertices $V(B_M)$ is the set of migrations in M, and each node/migration m in $V(B_M)$ is assigned a node-weight of $B(m)$. Consider a pair of migrations m_1, m_2 in M. The pair of vertices (m_1, m_2) are connected by an edge in B_M if and only if $B(\{m_1, m_2\})$ is non-zero, in which case we also assign a weight of $B(\{m_1, m_2\})$ to the edge (m_1, m_2).*

▶ **Lemma 5.5.** *At any stage of the G^* algorithm, a set O of migrations can be divided into disjoint subsets $O_1, O_2, \ldots, O_l$ such that $B(O) \leq \sum_{i=1}^{l} B(O_i)$.*

5.3 G* Iteration: Selecting an Optimal Same-Partition Set

We now design an algorithm to select the same-partition set of migrations with highest benefit-density, at a given stage/iteration of the G^* algorithm. Let us consider a stage of the G^* algorithm where a set of migrations have already been selected. Our goal is to pick a set $\mathcal{M}$ of same-partition migrations from the remaining migrations such that $B(\mathcal{M})/|\mathcal{M}|$ is maximized, where $B(\mathcal{M})$ is the benefit of $\mathcal{M}$ at the given stage of the G^* algorithm. Our overall algorithm of selecting an optimal $\mathcal{M}$ consists of the following steps.

1. For each given partition p_i, let M_i be the set of remaining (i.e., not yet selected by G^*) migrations that migrate a qubit to p_i. Note that each M_i is a maximal set of same-partition migrations.
2. For each M_i, we find a set $\mathcal{M}_i \subseteq M_i$ with highest benefit-density, as described later.
3. From the above $\mathcal{M}_i$'s, We pick the $\mathcal{M}_i$ with highest $B(\mathcal{M}_i)/|\mathcal{M}_i|$ as the optimal $\mathcal{M}$.

It is easy to see that the above returns an optimal same-partition set of migrations, as any same-partition set of migrations must be a subset of M_i for some i.

Selecting a Subset of M_i with Highest Benefit-Density. We now discuss how to select an optimal subset within a given M_i . We start with a lemma, which will help reduce the problem to that of selecting an optimal induced subgraph in the benefit graph of M_i.

▶ **Lemma 5.6.** *Let M be a same-partition set of migrations. Then, $B(M) = \sum_{m \in M} B(\{m\}) + \sum_{m_1, m_2 \in M} B(\{m_1, m_2\})$.*

The above lemma implies that finding the best subset within a given M_i is equivalent to finding an induced subgraph H in the benefit-graph of M_i that has the highest density of weight, i.e., maximum value of $(\sum_{e \in E(H)} w(e) + \sum_{v \in V(H)} w(v))/|V(H)|$. This is a weighted generalization of the well-studied *densest subgraph problem* which can be solved optimally in polynomial time. We discuss this below.

Weighted Densest Subgraph Problem. Given an unweighted graph G, the *densest subgraph problem* is to select an induced subgraph H in G such that $|E(H)|/|V(H)|$ is maximized. This problem can be solved optimally in polynomial time [6]. We are interested in the weighted generalization of this problem referred to as the *weighted densest subgraph* problem, wherein vertices and edges have (positive) weights associated, and the goal is to select an induced subgraph H with maximum $(\sum_{e \in E(H)} w(e) + \sum_{v \in V(H)} w(v))/|V(H)|$. The simple LP-based optimal algorithm as well as the more time-efficient 2-approximation algorithm for the unweighted version given in [6] can be both easily generalized to our weighted version of the problem. We briefly give both the algorithms, and defer the performance guarantee proofs (as they are simple generalizations of the proofs for the unweighted case in [6]).

LP-based Optimal Algorithm. The weighted densest subgraph problem can be easily represented as an ILP; the corresponding LP is as follows. Here, w_i is the weight of vertex i, and w_{ij} is the weight of edge (i, j) in the given graph.

$$\text{Maximize} \quad \sum_i y_i w_i + \sum_{ij} x_{ij} w_{ij}$$

$$\text{Subject to:} \quad (i)\ \ 0 \leq y_i, x_{ij} \leq 1; \quad (ii)\ \ x_{ij} \leq y_i; \quad (iii)\ \ \sum_i y_i \leq 1; \quad (iv)\ \ x_{ij} \leq y_j.$$

The optimal solution is obtained by: (i) Solving the above fractional LP; let the solution be $\{\overline{y_i}, \overline{x_{ij}}\ \}$. (ii) Picking a threshold τ appropriately and setting $\overline{y_i} = \lfloor \overline{y_i}/\tau \rfloor$. In (ii), we pick a threshold the LP objective; we do so by exhaustively trying all $\overline{y_i}$ values as the threshold. It can be shown by a simple generalization of the proof for the unweighted version [6] that the above process returns an optimal solution for the weighted densest problem.

2-Approximate Greedy Algorithm. A simple greedy algorithm to solve the weighted densest subgraph problem is to iteratively remove a vertex with the lowest sum of node weight and weight of incident edges, keep track of the resulting n subgraphs over n interations, and picking the best among them. This greedy algorithm can be easily shown to be 2-approximate, by a simple generalization of the proof for the unweighted version [6].

Overall G* Iteration. Algorithm 1 gives the pseudo-code of the overall algorithm for a single G^* iteration, which selects the same-partition set with highest benefit-density.

Algorithm 1 SINGLE ITERATION OF G^* ALGORITHM.

```
Input: The set of remaining migrations $M$, at a certain stage of $G^*$ Algorithm.
Output: A set of same-partition migrations $\mathcal{M}$ in $M$, with the highest benefit-density.

Let $M_1, M_2, \ldots, M_k$ be the disjoint and maximal same-partition subsets of $M$ corresponding to each of the partitions $p_1, p_2, \ldots, p_k$.
/* For each $M_i$, find a subset of $M_i$ with highest benefit-density */.
for all $i \leq k$ do
    Construct the benefit-graph $B(M_i)$ of $M_i$.
    Find the weighted densest subgraph $H_i$ in $B(M_i)$.
    $\mathcal{M}_i \leftarrow$ Vertices in $H_i$
end for
$\mathcal{M} \leftarrow$ The $\mathcal{M}_i$ with highest benefit-density.
return $\mathcal{M}$.
```

6 Evaluation

We now evaluate our algorithms empirically over random quantum circuits. The goal of our empirical study is to evaluate the performance of proposed techniques in terms of number of migrations incurred, i.e., the number of cat-entanglements, and compare them with the prior approach from [3].

Algorithms Compared. We compare our techniques with the algorithm proposed in [3], which is based on computing a min-cut in an appropriate hypergraph; we refer to this algorithm as `Martinez-19`. Our algorithms are all based on the Two-Step Algorithm of §3, with just different subroutines (from §4-5) for the second step. For the first-step of partitioning the qubits using a simple weighted graph, we use a third-party solver KaHyPar [1] and the refined weighted scheme discussed in §4. For the second-step, we use three different schemes and refer to the overall `DQC` algorithms as follows: (i) `Home-Cover` algorithm uses Algorithm 2 in the second step, (ii) G^*`_LP`, G^*`_Approx`, G^*`_Simple` algorithms use Algorithm G^* in the second step, with the LP-based, 2-approximation greedy, and simple greedy algorithms respectively for solving the weighted densest subgraph problem. The simple greedy algorithm to compute the weighted densest subgraph simply removes one vertex at a time to improve the weighted-density of the remaining graph, and stops when the remaining graph's weighted density can't be improved by removing any vertex. All our algorithms use the refined weights described in §3.2; usage of refined weights yielded a performance advantage of around 6% compared to using the simple weights.

Random Circuit Inputs and Parameter Values. We run our simulations over randomly generate quantum circuits. To generate quantum circuit instances, we vary the following parameters: number of qubits, total number of gates per qubit, fraction of gates that are binary (CZ) gates, and number of given partitions. We use an imbalance-factor ν of 1.1 for all the experiments. In the below plots, we vary one of the parameters and fix the remaining three to their default values. The default value for number of qubits is 50, total number of gates per qubit is 50, and number of partitions is 10. For the fraction of binary gates, we use two default values: 50% and 80%, as this parameter has a strong impact on performance of algorithms and in practice, the fraction of binary gates can be high.[2] Each data point in the below experiments is obtained from an average of 5 different random instances.

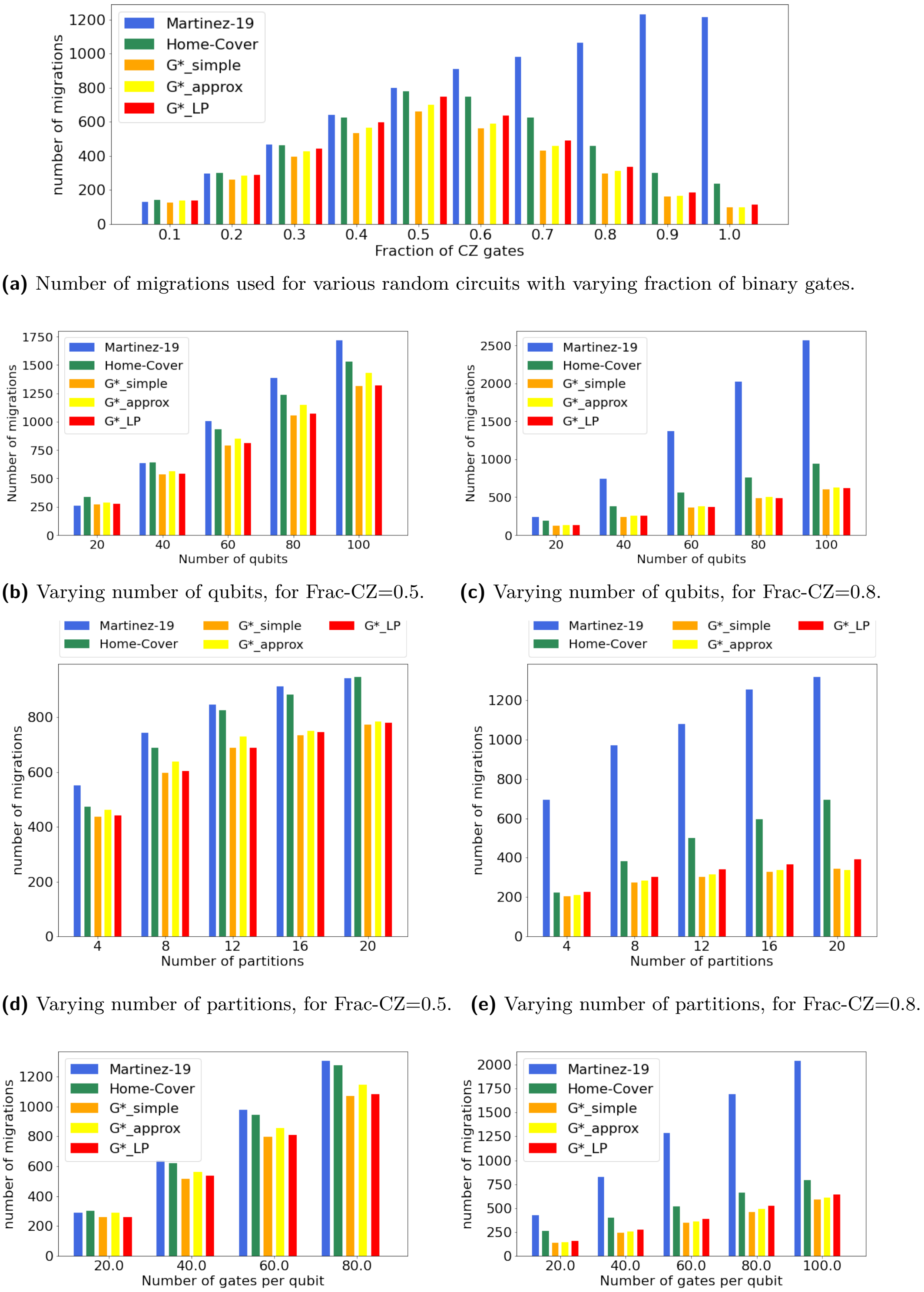
Evaluation Results for Varying Parameters. We present our results in Figures 4b-4f. In general, we observe that all the three G^*-based algorithms performing similarly and significantly outperforming the `Martinez-19`, across all our experiments. In particular, the G^*-based algorithms perform up to 90% better (i.e., incurring merely 20% of the migrations/cat-entanglements used by `Martinez-19`; see Fig. 4a). The `Home-Cover` algorithm, which implicitly conserves execution memory usage at any single partition, also performs up to 80% better than `Martinez-19`. Among the G^*-based algorithms, surprisingly the G^*`_Simple` performs slightly better than the other two; note that, this does not contradict the optimality of LP-based algorithm for the densest weighted subgraph, since densest weighted subgraph solution is only used within an iteration of the G^* algorithm.

Figure 4a plots results for varying fraction of binary (CZ) gates in the circuit. We observe that the performance of all our algorithms, unlike `Martinez-19`, improves significantly with increasing fraction of binary gates; this can be attributed to the fact that higher fraction increases the number of gates covered by a single migration, and our algorithms are able to take better advantage of it. Figures 4b and 4c show the performance of various algorithms for increasing number of qubits, with 50% and 80% CZ gates respectively, while using default values for other parameters. The performance of our algorithms in comparison to `Martinez-19` improves with increasing number of qubits. Figures 4d and 4e plot results for varying number of partitions. Here, we observe that performance of `Home-Cover` wrt G^*-based algorithms worsens with increase in partitions; this may be due to the fact that `Home-Cover`'s restriction of home-partition execution becomes more pronounced with increase in number of partitions. Finally, Figures 4f and 4g plot the performance of algorithms for varying number of gates per qubit. We discuss execution memory usage in Appendix A.8.

7 Conclusions

In this paper we considered the problem to distribute a quantum circuit over a network of QCs, such that the communication cost minimized. We presented a two-step heuristic that outperforms prior techniques. In future work, we plan to look at various generalizations of the problem, e.g., for nodes with non-uniform capacities, links with non-uniform communication costs, constraining the execution memory at each partition, allowing multiple modes of communications (e.g., teleportation as well as cat-entanglement), allowing for unary gates to be executed at any partition (i.e., allowing for dynamic home-partition of a qubit). In addition, we plan to investigate better heuristics for the first step of our algorithm.

[2] E.g., the Quantum Fourier Transform (QFT) circuit has has $O(n^2)$ controlled-phase gates and $O(n)$ other gates; since the controlled-phase gates can be treated like CZ gates, the fraction of binary gates in a QFT circuit can be arbitrarily high. Also, when we convert binary gates in an arbitrary circuit to CZ gates, the fraction of binary gates is expected to be 50% as a binary gate yields a CZ gates flanked by unary gates (and long runs of unary gates can be considered as a single unary gate, in our context).



(a) Number of migrations used for various random circuits with varying fraction of binary gates.

(b) Varying number of qubits, for Frac-CZ=0.5.

(c) Varying number of qubits, for Frac-CZ=0.8.

(d) Varying number of partitions, for Frac-CZ=0.5.

(e) Varying number of partitions, for Frac-CZ=0.8.

(f) Varying gate ratio, for Frac-CZ=0.5.

(g) Varying gate ratio, for Frac-CZ=0.8.

Figure 4 Performance of various algorithms for varying fraction of CZ gates ((a)), varying number of qubits ((b)–(c)), varying number of partitions ((d)–(e)), and varying number of gates per qubit ((f)–(g)). Above, Frac-CZ (fraction of CZ gates) is 0.5 in (b), (d), and (f), and is 0.8 in (c), (e), and (g).

References

1 Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct k-way hypergraph partitioning algorithm. In *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2017.

2 Konstantin Andreev and Harald Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6), 2006.

3 P. Andres-Martinez and C. Heunen. Automated distribution of quantum circuits via hypergraph partitioning. *Phys. Rev. A*, 100(3), 2019.

4 C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein–Podolsky–Rosen channels. *Phys. Rev. Lett.*, 70(13), 1993.

5 M. Caleffi, A. S. Cacciapuoti, and G. Bianchi. Quantum internet: from communication to distributed computing!, 2018.

6 Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*. Springer, 2000.

7 J. I. Cirac, A. K. Ekert, S. F. Huelga, and C. Macchiavello. Distributed quantum computation over noisy channels, phys. rev. a. *Phys. Rev. A*, 59(4249), 1999.

8 O. Daei, K. Navi, and M. Zomorodi-Moghadam. Optimized quantum circuit partitioning. *Int. J. Theor. Phys.*, 59(12):3804–3820, 2020.

9 D. Dieks. Communication by EPR devices. *Physics Letters A*, 92(6), 1982. `doi:10.1016/0375-9601(82)90084-6`.

10 L.-M. Duan, M. D. Lukin, J. I. Cirac, and P. Zoller. Long-distance quantum communication with atomic ensembles and linear optics. *Nature*, 414(6862), November 2001. `doi:10.1038/35106500`.

11 J. Eisert, K. Jacobs, P. Papadopoulos, and M.B. Plenio. Optimal local implementation of non-local quantum gates. *Phys. Rev. A*, 62(052317-1), 2000.

12 B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2), 1970.

13 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. `doi:10.1017/CBO9780511976667`.

14 Qiskit. `https://qiskit.org/`.

15 Quipper. `https://www.mathstat.dal.ca/~selinger/quipper/doc`.

16 Nicolas Sangouard, Christoph Simon, Hugues De Riedmatten, and Nicolas Gisin. Quantum repeaters based on atomic ensembles and linear optics. *Reviews of Modern Physics*, 2011.

17 W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886), 1982. `doi:10.1038/299802a0`.

18 A. Yimsiriwattana and S. J. Lomonaco Jr. Distributed quantum computing: A distributed Shor algorithm. *Quantum Information and Computation II*, 5436, 2004.

19 A. Yimsiriwattana and S. J. Lomonaco Jr. Generalized GHZ states and distributed quantum computing. *AMS Cont. Math.*, 381(131), 2005.

A Appendix A

A.1 Illustrating Lemma 4.2 over Running Example

See Figure 5.

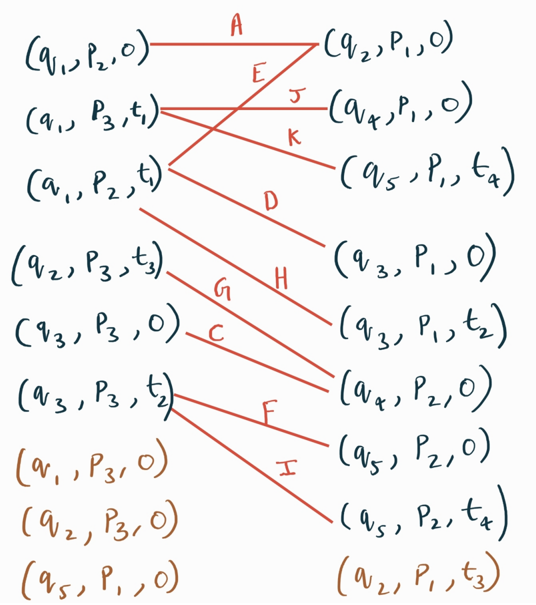


Figure 5 The G_{HC} graph connecting migrations with the home-covered binary gates, for the partitioned circuit of the running example. Here, each binary gate is represented by an edge that connects the two migrations (represented as triplets) that home-cover it.

A.2 Proof of Lemma 4.2

Proof. Consider a arbitrary binary gate $g = (q_i, q_j, t)$. Let t_i (t_j) be timestamp of the last unary gate on q_i (q_j) before t or 0 if there is no unary gate on q_i (q_j) before t. Then, its easy to see by Def. 4.1 that the only migrations that home-cover g are $(q_i, \pi(q_j), t_i)$ or $(q_j, \pi(q_i), t_j)$. ◀

A.3 Proof of Lemma 4.3

Proof. Consider a cycle of length n in G_{HC}, and let the i^{th}vertex in the cycle be the migration (q_i, p_i, t_i) for $0 \leq i \leq n-1$. We claim that $\pi(q_i) = \pi(q_{(i+2) \mod n})$ for $0 \leq i \leq n-1$. Consider the sequence of vertices $(q_i, p_i, t_i), (q_{(i+1) \mod n}, p_{(i+1) \mod n}, t_{(i+1) \mod n})$, and $(q_{(i+2) \mod n}, p_{(i+2) \mod n}, t_{(i+2) \mod n})$. Since $(q_{(i+1) \mod n}, p_{(i+1) \mod n}, t_{(i+1) \mod n})$ is connected to both the other vertices, we have that $p_{(i+1) \mod n} = \pi(q_i)$ and $p_{(i+1) \mod n} = \pi(q_{(i+2) \mod n})$ which implies that $\pi(q_i) = \pi(q_{(i+2) \mod n})$. The above implies that for an odd-length cycle, we'll get $p_i = \pi(q_i) = p$ for all i and some particular partition p, which is impossible for a migration/vertex (see Def. 3.3). ◀

A.4 Optimal MS-HC Algorithm Psuedo-Code

Algorithm 2

Input: A partitioned quantum circuit. Let T be the set of non-local gates.
Output: An optimal set of migrations that home-covers all the non-local gates T.

```
$M \leftarrow \emptyset$
for all $g \in T$ do
    $M \leftarrow M \bigcup \{\text{migrations that home-cover } g\}$.
end for
Construct the bipartite graph $G_{HC}$ over $T$ and $M$.
$M^* = \textsc{MinVC}(G_{HC})$.
return $M^*$.
```

A.5 Proof of Theorem 5.3

Proof. Let $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_k$ be the sets of migrations picked by the Greedy Algorithm over k iterations. Let a_i is the number of gates covered by $\mathcal{M}_i$, that weren't previous covered by $\mathcal{M}_1$ to $\mathcal{M}_i$ sets. Thus, the overall greedy solution covers $(a_1 + a_2 \ldots + a_k)$ gates, which is equal to N, the total number of gates in the circuit (as the greedy algorithm only terminates when all the gates have been covered). Let be optimal solution be O; here, O is a set of migrations that cover all the N gates in the circuit.

Consider a stage when the Greedy Algorithm has already selected $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_{i-1}$ sets of migrations. We can observe the following.

- The total number of gates already covered by the greedy sets $\mathcal{M}_1$ to M_{i-1} is $\sum_{j=1}^{i-1} a_j$. Thus, the number of gates covered by the optimal solution that have not been yet covered by the greedy sets $\mathcal{M}_1$ to $\mathcal{M}_{i-1}$ is at least $N - \sum_{j=1}^{i-1} a_j$. This follows from "monotonicity" of the coverage function; in particular, from the fact that the $\mathcal{M}_1$ to M_{i-1} sets and the optimal solution together still cover N gates.
- By Lemma 5.5, the optimal set O can be divided into disjoint same-partition subsets of migrations $O_1, O_2, \ldots, O_m$ such that $B(O) \leq \sum_{i=1}^{i=m} B(O_m)$ where $B(X)$ is the benefit of set X at this stage.
- By pigeon hole principle and above, there exists a same-partition set O_l of migrations such that $B(O_l)$ at this stage is at least

$$|O_l|(N - \sum_{j=1}^{i-1} a_j)/|O|.$$

- Since the next set $\mathcal{M}_i$ picked by the Greedy Algorithm is a same-partition set of migrations with the highest benefit-density, i.e., one that covers the most number of uncovered (by $\mathcal{M}_1$ to $\mathcal{M}_{i-1}$) gates per unit migration, the number of new gates covered by $\mathcal{M}_i$ is at least $|\mathcal{M}_i|(N - \sum_{j=1}^{i-1} a_j)/|O|$. Thus, we have

$$a_i \geq |\mathcal{M}_i|(N - \sum_{j=1}^{i-1} a_j)/|O|.$$

Now, using the above equation, it is easy to show by induction that $(N - \sum_{j=1}^{i} a_j) \leq N(1 - 1/|O|)^{i'}$, where $i' = \sum_{j=1}^{i} |\mathcal{M}_j|$, the total number of migrations in the Greedy solution till the i^{th}stage. Thus, when $i' = |O| \ln N$, we get $(N - \sum_{j=1}^{i} a_j)$ (the number of uncovered elements) as less than 1, which is when the Greedy Algorithm stops (after one more step). Thus, the number of migrations selected by the Greedy solution is we $|O| \ln N$, showing that the Greedy Algorithm is $(\ln N)$-factor approximation algorithm, where N is the total number of gates in the circuit. ◀

A.6 Proof of Lemma 5.5

Proof. Let O be the optimal solution of the given instance of the MS-GC problem. Let B_O be the benefit-graph of O. We make the following two claims. **First**, we claim that each connected component in the benefit graph B_O is a same-partition set. This claim follows from the observation that if two nodes are *connected* by a path in the benefit graph B_O, then the corresponding migrations migrate the appropriate qubits to the *same* partition. Recall that $B(\{m_1, m_2\})$ is non-zero if and only if m_1 and m_2 are migrating the corresponding qubits to the same partition. **Second,** let $B_{O1}, B_{O2}, \ldots, B_{Ol}$ be the l connected components of B_O, with O_1 to O_l denoting the subsets of migrations corresponding to the connected components. We claim that $B(O) \leq \sum_{i=1}^{l} B(O_i)$. This follows[3] from the following facts: (i) Each O_i is a same-partition set of migrations (from the first claim above), and (ii) If a gate g is covered by migrations in O, then it is covered by a single or a pair of migrations in some O_i.[4] Thus, the lemma follows. ◀

A.7 Proof of Lemma 5.6

Proof. This follows from the fact that if a non-local gate g is covered by some migration(s) in M, then *only one* of the following is true: (i) there is a *unique* migration $m \in M$ that covers g, or (ii) there is a *unique* pair of migrations $\{m_1, m_2\}$ that together cover g. ◀

A.8 Evaluating Execution Memory vs. Communication Trade-off

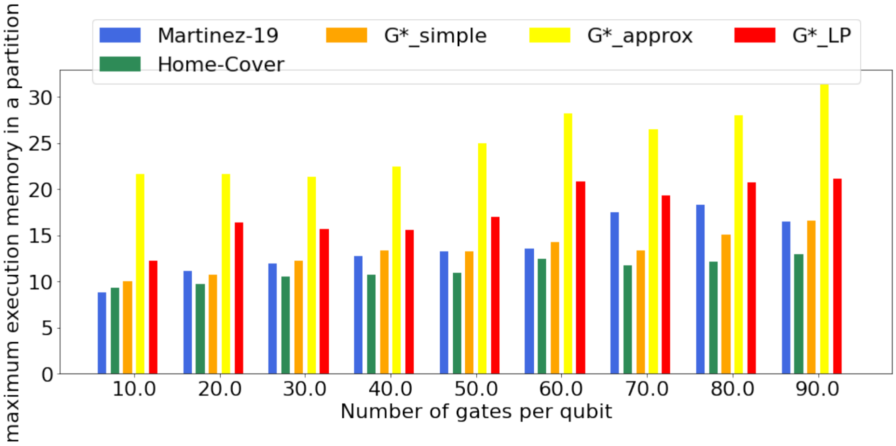


Figure 6 Maximum ebit memory required in a single partition for varying number of gates per qubit.

[3] We note that, in general, $B(O)$ may not be equal to $\sum_{i=1}^{l} B(O_i)$, since a gate may be independently covered by multiple (individual or pairs of) migrations across different O_i's.

[4] Note that a gate g is never covered by a pair of migrations together that lie in different O_i's.

Recall that our Home-Cover algorithm restricts execution of binary gates to the home-partition of one of the operands, to intuitively minimize the execution memory (i.e., maximum number of linked copies migrated from other partitions, at any instant) usage in any partition. To verify this intuition, we plot in Fig. 6, the maximum execution memory usage across partitions, and observe a modest difference in the usage of execution memories between Home-Cover and G^*_Simple algorithms but a higher usage in other G^*-based algorithms. Also, observe that the memory usage of Martinez-19 is slightly higher than that of G^*_Simple.

Game Theoretical Framework for Analyzing Blockchains Robustness

Paolo Zappalà ✉
Orange Labs, 92320 Chatillon, France
LIA, Avignon Université, 84029 Avignon, France

Marianna Belotti ✉
BDTD 60, Caisse des Dépôts, 75013 Paris, France
Cedric, Cnam, 75003 Paris, France

Maria Potop-Butucaru ✉
Lip6, CNRS UMR 7606, Sorbonne University, 75005 Paris, France

Stefano Secci ✉
Cedric, Cnam, 75003 Paris, France

Abstract

In this paper we propose a game theoretical framework in order to formally characterize the robustness of blockchains systems in terms of resilience to rational deviations and immunity to Byzantine behaviors. Our framework includes necessary and sufficient conditions for checking the immunity and resilience of games and an original technique for composing games that preserves the robustness of individual games. We prove the practical interest of our formal framework by characterizing the robustness of various blockchain protocols: Bitcoin (the most popular permissionless blockchain), Tendermint (the first permissioned blockchain used by the practitioners), Lightning Network, a side-chain protocol and a cross-chain swap protocol. For each one of the studied protocols we identify upper and lower bounds with respect to their resilience and immunity (expressed as no worse payoff than the initial state) face to rational and Byzantine behaviors.

2012 ACM Subject Classification Networks

Keywords and phrases Blockchain protocols, Distributed algorithms, Game-theoretical modeling, Fault tolerance, Failure robustness

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.42

Related Version *Full Version*: `https://hal.archives-ouvertes.fr/hal-02634752/document`

1 Introduction

Distributed Ledger Technologies (DLTs) allow sharing a ledger of transactions among multiple users forming a peer-to-peer (P2P) network. DLTs characterized by a block architecture are called "blockchains". They enable users to transfer cryptoassets in a decentralized manner by means of modular protocols adopted by the users themselves. Beyond the traditional blockchain architectures (*layer-1 protocols*), the literature proposes other protocols that respectively define and regulate interactions in an overlaying network (*layer-2 protocols*) and interactions between different blockchains (*cross-chain protocols*). Each of these protocols establishes the instructions users must follow in order to interact with or through a blockchain. In a blockchain system players can be classified, as proposed in [2] for classical distributed systems, in three different categories: (i) players who follow the prescribed protocol i.e., *altruistic*, (ii) those who act in order to maximise their own benefit i.e., *rational*, and (iii) players who may deviate arbitrarily from the prescribed protocol, i.e. *Byzantine* (cf. [18]). Interactions among users are usually modeled with game theory which analyzes the decision-making process in presence of multiple rational agents, called *players* or *agents*.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 42; pp. 42:1–42:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the context of blockchain systems, game theoretical frameworks were introduced in [36, 37] to analyze security aspects and incentive compatibility of Nakamoto's consensus protocol (i.e., Proof-of-Work [25]) characterizing the very first blockchain implementation known as Bitcoin. Users participating in the consensus mechanism (i.e., miners) are considered as individually rational moved by the mere intention to increase their revenues i.e., the rewards earned form the mining activities [7, 32]. Authors in [8, 13, 35, 12] adopt different utility functions for miners and pools that consider costs and relative rewards. Concerning layer-2 and cross-chain protocols, game theoretical analysis are carried out by [5, 6, 15]. These analyses are strictly specific to the particular deployment context rather than to a generic blockchain. Most of the game theoretical models adopted to design secure and robust blockchain protocols, surveyed in [23], (i) address protocols characterizing specific blockchain implementations, (ii) analyze miners' behaviours in the consensus phase and (iii) adopt Nash Equilibria as solution concept.

Concerning *rational agents*, the existing analyses include the study of equilibria and the evaluation of their properties. The most studied and adopted solution concept in literature is the Nash Equilibrium, i.e., a strategy profile in which no player has interest in individually deviating from her own strategy. A first approach to the analysis of robustness is to compare Nash Equilibria, through indices such *Price of Byzantine Anarchy* [24], *Price of Malice* [24] and *Price of Anarchy* [20]. This approach summarizes the outcomes of the games representing protocols, but it does not show explicitly the implementation risks of such systems. A second approach is to analyze peculiar Nash Equilibria. Authors of [28] take probability into account and extend the concept of Nash Equilibrium. In [18], *virtual utility* – alternative to the classical game utility – is introduced to capture the blockchain agreement structure. The analysis of robustness with respect to *Byzantine agents* was modeled in [3] with a Bayesian game. The authors provide the analysis of Tendermint protocol [22]. This method allows making forecasts on the expected outcomes of a game, but it does not provide a comprehensive analysis of the risks. It should be noted that none of the previous works is generic enough to propose a methodology for analyzing the robustness of blockchain protocols to both rational and Byzantine players.

The first generic framework for analyzing the *robustness of distributed protocols* with respect to the behavior of rational and Byzantine players was proposed by the authors of [1] who introduced the concept of *mechanism* (i.e., a pair game-prescribed strategy). Moreover in [1] authors introduced the notions of (i) k-resilience, (ii) practicality and (iii) t-immunity. A strategy profile is defined as *k-resilient* if there is no coalition with at most k players having an incentive to deviate from the prescribed protocol. The category of *practical* strategy profiles is defined when equilibria with weakly dominated strategies are excluded. Finally, *t-immunity* denotes a situation where no player gets a lower outcome if there are at most t Byzantine players that can play any possible strategy. Interestingly, despite its mathematical beauty this framework was never used to analyze the robustness of blockchain protocols.

Our contribution. In this paper we follow the line of work opened in [1] and present a game theoretical framework aiming at characterizing the *robustness of blockchain protocols*. Our contributions can be summarized as follows: (a) we prove that t-immunity property defined in [1] is not verified by a large class of blockchain protocols (cf. Table 1). It should be noted that the authors of [1] already observed that "t-immunity is often impossible to be satisfied by practical systems" and left open the definition of a weaker property; (b) we introduce the new concept of *t-weak-immunity*; a mechanism is t-weak-immune if any altruistic player receives no worse payoff than the initial state, no matter how any set of

t players deviate from the prescribed protocol. This new concept is sufficiently strong to capture the robustness of a large class of blockchain protocols (cf. Table 1); (c) we identify and prove necessary and sufficient conditions for a mechanism to be k-resilient and t-weak-immune; (d) we define a new operator for game composition and prove that it preserves the robustness properties of the individual games; (e) we use our generic framework and the composition operator we study the robustness of a representative set of layer-1, layer-2 and cross-chain protocols: Tendermint [22], Bitcoin [25], Lightning Network protocol [30], the side-chain protocol [29] and the very first implementation of a cross-chain swap protocol proposed in [27] and formalized in [15].

For each one of the analyzed protocols we provide bounds on the number of Byzantine processes in order to verify t-weak immunity. Furthermore, for the same class of protocols we compute bounds on the number of rational processes in order to achieve k-resilience. Our results are reported in Table 1. Interestingly, our analysis allowed us to spot the weakness of the Lightning Network protocol [30] to Byzantine behaviour. Therefore, we propose and further analyze an alternative version of the protocol.

The paper is structured as follows. Section 2 is devoted to the definition of mechanism, (k, t)-robustness, necessary and sufficient conditions for optimal resilience and weak immunity and, composition of mechanisms. We apply in Section 3 the methodology developed in Section 2 to prove the robustness of the protocols presented in [25, 22, 30, 29, 27]. Section 4 concludes the paper. All the illustrations of the models as well as all the proofs for the results presented in this paper are available at [38].

Table 1 Immunity and resilience properties for Tendermint [22], Bitcoin [25], Lightning Network [30], a side-chain protocol [29] and a cross-chain swap protocol [27, 15] with respect to the number of rational deviating agents (k) and the number of Byzantine deviating agents (t) where n is the total number of players in the game.

Protocol	k-resilience	t-immunity	t-weak immunity	Results
Tendermint	**Yes**, k $< n/3$	**No**	**Yes**, t $< n/3$	Thm. 8
Bitcoin	**Yes**, k $< 3n/20$	**No**	**No**	Thm. 10
Lightning Network	**Yes**, k $< 3n/20$	**No**	**No**	Thm. 12
Closing module	Yes	No	No	Thm. 17
(Alternative closing module)	(Yes)	(No)	(Yes)	Thm. 18
Other modules	Yes	No	Yes	Thm. 14, 15, 20, 22, 23
Side-chain (Platypus)	**Yes**, k $< n/3$	**No**	**Yes**, t $< n/3$	Thm. 25
Cross-chain Swap	**Yes**	**No**	**Yes**	Thm. 28

2 Games theoretical framework for Analyzing protocols robustness

2.1 Preliminaries on Game Theory

The basic idea of a game is to capture a set of players which may act sequentially or simultaneously (cf. [21] for more details). The theoretical concept adopted in this paper is the one of *extensive form game*. A game of this type is represented formally by a tuple $\Gamma = \langle N, T, P, (A_h)_{h\in V}, (u_i)_{i\in N}\rangle$ where N is the set of players, $T = (V, E)$ is a directed rooted tree, $Z \subset V$ is the set of terminal nodes, $P : V \setminus Z \to N$ is a function assigning to each non-end node a player in N, $A_h = \{(x_h, x_i) \in E\}$ for each node $h \in V \setminus Z$ is the set of edges going from node h to some other nodes and represents the set of actions at node h of the tree T, $\Omega_i = \{s_i : V \setminus Z \to A_1 \times A_2 \times \dots A_h \times \cdots \times A_H, h : P(h) = i\}$ is the set of pure strategies of player i, $\mathcal{S}_i = \{\sigma_i : \Omega_i \rightarrow [0,1], \sum_{s \in \Omega_i} \sigma_i(s) = 1\}$ is the set of mixed strategies of player i and $u_i : Z \rightarrow \mathbb{R}$ is the utility function for player $i \in N$. Every pure strategy of player i is a function that assigns an action $a \in A_h$ to every node $h \in V \setminus Z$ in which player i is involved (formally, $h : P(h) = i$). A mixed strategy is a probability distribution over the set of pure strategies of player i. For the sake of simplicity in the notation, analyses and proofs will involve pure strategies only, as the results can be easily generalized then for general mixed strategies. Every game in extensive form can be reformulated in a more compact way (i.e., *normal form*) with a tuple $\Gamma = \langle N, \mathcal{S}, u \rangle$, in which the set of players $N = \{1, \ldots, n\}$ denotes the players involved in the protocol, $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2 \times \cdots \times \mathcal{S}_n$ where $\mathcal{S}_i$ is the set of strategies of player i and $u : \mathcal{S} \rightarrow \mathbb{R}^n$ is the utility function of the players. Each player can pick her own strategy $\sigma_i \in \mathcal{S}_i$ generating a strategy profile $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_i, \ldots, \sigma_n) \in \mathcal{S}$ and a utility vector $u(\sigma)$ whose i-th component denotes the utility for player i.

A *solution concept* $\sigma \in \mathcal{S}$ is a strategy profile such that the outcome $u(\sigma)$ pleases every player so that they have no incentive in changing their strategy σ_i. The most known solution concept is the *Nash Equilibrium*, where no player has an incentive to unilaterally change strategy [26]. Formally, a strategy profile σ is a Nash Equilibrium if $u_i(\sigma_1, \sigma_2, \ldots, \sigma_i, \ldots, \sigma_n) \geq u_i(\sigma_1, \sigma_2, \ldots, \tau_i, \ldots, \sigma_n)$ for every player i and for every $\tau_i \in \mathcal{S}_i$. Nash [26] proved that every game in normal form admits at least one Nash Equilibrium. A Nash Equilibrium $\sigma \in \mathcal{S}$ is said to be *strong* if and only if for all $C \subseteq N$ and all $\tau_C \in \mathcal{S}_C$, there exists $i \in C$ such that $u_i(\sigma_C, \sigma_{-C}) \geq u_i(\tau_C, \sigma_{-C})$ i.e., given the strategy of its complements as given, no coalition can deviate in a way that benefits all of its members [9]. Strong Nash Equilibria are easy to be identified but they do not always exist. A Nash Equilibrium is said to be *stable* if it is still a Nash Equilibrium after small deviations in the game [16]. Moreover, as stated in [19], there always exists a stable Nash Equilibrium. Stable Nash Equilibria survive after the iterated deletion of *weakly dominated strategies*, i.e., those strategies $\sigma_i \in \mathcal{S}_i$ that perform as well as or worse than another strategy $\sigma'_i \in \mathcal{S}_i$ no matter which strategy the other players choose (formally, we have that $u_i(\sigma_i, \tau_{-i}) \leq u_i(\sigma'_i, \tau_{-i})$ for all $\tau_{-i} \in \mathcal{S}_{-i}$). In the process of *iterated deletion*, weakly dominated strategies are excluded from the set of strategies available to players and the set of Nash Equilibria is recomputed.

In the paper we analyze protocols modeled as games by studying the strategy profile associated to rational and Byzantine players; we identify the Nash Equilibria of the games and assign some properties to the respective strategy profiles.

2.2 Mechanisms and Robustness

The paper analyzes blockchain protocols in which players can either decide to follow or not the prescribed instructions. The aim of the paper is to model these problems and understand whether the players are incentivized to follow or deviate from the prescribed protocol being respectively altruistic or Byzantine agents. In the following (i) we recall and extend the game theoretical framework based on the concept of mechanism and its properties, (ii) we define new properties on protocol robustness and (iii) we study properties interdependence.

Let us consider a game in normal form $\Gamma = \langle N, \mathcal{S}, u \rangle$ where players find themselves in an initial state, i.e., before starting the application of the protocol. We assign $u_i(\sigma) = 0$ for every $\sigma \in \mathcal{S}$ when the player i is indifferent between the outcome of the strategy profile σ and the initial state one. Analogously, we assign positive utility, $u_i(\sigma) > 0$, when the outcome of σ corresponds to the final state provided by the protocol and negative utility, $u_i(\sigma) < 0$, when the outcome of σ is worse than the initial state one. The values of u_i, for all $i \in N$, correspond to the marginal utility with respect to the initial state. Every decision-making problem is modeled by a game $\Gamma = \langle N, \mathcal{S}, u \rangle$, which shows all the possible strategies available to the players, including following the prescribed protocol and all its possible deviations. A specific protocol consists of a strategy profile $\sigma = (\sigma_1, \ldots, \sigma_n) \in \mathcal{S}$ and it is denoted by a pair (Γ, σ), called *mechanism* [1]. Every player i is advised to play strategy $\sigma_i \in \mathcal{S}_i$ i.e., the recommended strategy σ is the prescribed protocol. Evaluating the robustness to deviations of a distributed protocol corresponds to identifying the properties of the mechanism (Γ, σ). Players can decide to deviate for two different reasons. On one hand, they can cooperate in order to find a strategy profile that provides a better outcome than the one given by the protocol. On the other hand, some players can behave maliciously for no specific reason and harm the altruistic ones. These two behaviours are prevented, according to [1], if prescribed distributed protocols are respectively (i) practical and k-resilient and/or (ii) t-immune.

A mechanism (Γ, σ) is *practical* if σ is a Nash Equilibrium of the game Γ after the iterated deletion of weakly dominated strategies. Players have a very low incentive to play weakly dominated strategies since they have available a different strategy providing no lower outcome in any scenario. If a mechanism is practical, these strategies are not played.

A mechanism (Γ, σ) is *k-resilient* if there is no coalition of at most k players having an incentive to simultaneously change strategy to get a better outcome. Formally, a strategy profile $\sigma \in \mathcal{S}$ is a *k-resilient equilibrium* if for all $C \subseteq N$ with $1 \leq |C| \leq k$, all $\tau_C \in \mathcal{S}_C$ and all $i \in C$, we have $u_i(\sigma_C, \sigma_{-C}) \geq u_i(\tau_C, \sigma_{-C})$. The concept of k-resilience denotes the tendency of a set of k players to cooperate to move to an equilibrium that differs from the prescribed one. Hence k-resilience generalizes the concept of Nash Equilibrium.

A mechanism (Γ, σ) is *t-immune* if, given at most t players choosing any strategy different from the prescribed one, the rest of the players receive at least the utility they would get if everyone followed the protocol. Formally, a strategy profile $\sigma \in \mathcal{S}$ is *t-immune* if for all $T \subseteq N$ with $|T| \leq t$, all $\tau_T \in \mathcal{S}_T$ and all $i \in N \setminus T$, we have $u_i(\sigma_{-T}, \tau_T) \geq u_i(\sigma)$. The property of t-immunity is very strong and difficult to satisfy since it requires that the protocol provides the best outcome no matter how a set of t players deviates. We therefore introduce a weaker version of the property – *t-weak-immunity* – guaranteeing that non deviating players receive at least the utility value of the initial state (i.e., players receive a positive outcome).

▶ **Definition 1** (t-weak-immunity). *A mechanism (Γ, σ) is* t-weak-immune *if for all $T \subseteq N$: $|T| \leq t$, all $\tau_T \in \mathcal{S}_T$ and all $i \in N \setminus T$, we have $u_i(\sigma_{-T}, \tau_T) \geq 0$.*

A player that joins a t-weak-immune mechanism will not suffer any loss (i.e., outcome with negative utility) if there are at most t deviating players in the game. We say that a mechanism is *weak immune* if it is t-weak-immune for all $t \in N$ and that a mechanism is *(k, t)-robust* if it is k-resilient and t-weak-immune.

Following the terminology introduced in [1], if every strict subset of players has no incentive to change their strategy σ, we say that the mechanism (Γ, σ) is *strongly resilient*. The concepts of k-resiliency and practicality are strictly connected with the properties of Nash Equilibria, such as *strength* [9] and *stability* [16, 19], which have been fully studied in [11]. Indeed, it is possible to prove that (i) if a mechanism (Γ, σ) is strongly resilient, then σ is a strong equilibrium of Γ and that (ii) if σ is a stable equilibrium of Γ, then the mechanism (Γ, σ) is practical (cf. [38] for details). In [16], authors prove that there always exists at least one stable Nash Equilibrium, therefore as a corollary there is always at least one practical mechanism. We know from [19] that the properties of strength and stability are independent. This means that we cannot draw conclusions about a property knowing whether the other property is fulfilled or not. Thus, we can state the following theorem.

▶ **Theorem 2** (independence). *The property of strongly resiliency and practicality are independent.*

We say that a mechanism (Γ, σ) is *optimal resilient* if it is practical and strongly resilient. In the sequel we verify whether protocols can be modeled with strongly resilient and/or practical mechanisms. Both properties have to be verified since they are independent. If a protocol does not provide a mechanism with a strong Equilibrium, it is necessary to compute k such that k-resiliency is fulfilled. On the other hand, given a generic game Γ, it is always possible to easily identify which are the practical mechanisms that always exist.

2.3 Composition of Games and Mechanisms

Blockchains systems are complex protocols designed in a modular way. In order to study the robustness of such complex protocols, we need to analyze the individual modules and infer the properties of the system by composition. For this scope we introduce the new notion of *composition of games* that, to the best of our knowledge, has never been defined in the literature. Given two different games A and B, the composition of games is defined by the operator $\odot$, hence $A \odot B$ denotes the composition of game A and B. Given two games that are played separately and independently, the composition corresponds to players picking a strategy from each game and receiving as utility the sum of the utilities of the two games.

▶ **Definition 3** (games composition). *Given $A = \langle N, \mathcal{S}_A, u_A \rangle$ and $B = \langle N, \mathcal{S}_B, u_B \rangle$ two games in normal form with the same set of players N, two different sets of strategies $\mathcal{S}_A = \{\mathcal{S}_{Ai} : i \in N\}$ and $\mathcal{S}_B = \{\mathcal{S}_{Bi} : i \in N\}$ and two different utility functions: $u_A : \mathcal{S}_A \to \mathbb{R}^N$ and $u_B : \mathcal{S}_B \to \mathbb{R}^N$ then, it is possible to define a new game $C = A \odot B$, called composition of A and B, characterized as follows: $C = \langle N, \mathcal{S}_C, u_C \rangle$, where N is the set of the players, $\mathcal{S}_C := \{(s_{Ai}, s_{Bi}), s_{Ai} \in \mathcal{S}_{Ai}, s_{Bi} \in \mathcal{S}_{Bi}, \forall i \in N\}$ is the set of strategies and $u_C(\{(\sigma_{Ai}, \sigma_{Bi})\}) := u_A(\{\sigma_{Ai}\}) + u_B(\{\sigma_{Bi}\})$ is the utility function.*

In the context of non-cooperative games linear transformations of utility functions are considered invariant transformations since they preserve the main properties of the game [14]. Therefore, we define the utility function of the composition of games as the sum of the utility functions of the composed games. It is possible to extend the definition of games composition to pairs of games in which different sets of players are involved. Indeed, if a player i is involved in game A but not in game B, it is possible to extend game $B = \langle N, \mathcal{S}_B, u_B \rangle$ to $B' = \langle N', \mathcal{S}'_B, u'_B \rangle$ in which player i is added ($N' = N \cup \{i\}$) and she is assigned a "null" strategy ($\mathcal{S}'_B = \mathcal{S}_B \times \{\sigma_\emptyset\}$) not influencing the utilities of the outcomes. Formally, for all $s \in \mathcal{S}_B$ and for all $j \in N' \setminus \{i\}$ we have that $u'_j(s, \sigma_\emptyset) = u_j(s)$, while for $i \in N'$ we have that $u_i(s, \sigma_\emptyset) = 0$. Intuitively it is possible to extend the definition of games composition to more than two games. In Section 3.3.4 we use the notation $A \odot B \odot C$ to represent either game $A \odot (B \odot C)$ or $(A \odot B) \odot C$. The following propositions allow us to (i) model the building blocks of complex protocols, (ii) study the properties of the subsequent mechanisms and (iii) deduce the properties of the composed protocol through the composition of mechanisms.

Concerning the solutions of the composition of games, we prove that Nash Equilibria can be identified by selecting equilibria within the single games. It is not possible to create or destroy Nash equilibrium strategies by composing independent games.

▶ **Theorem 4** (Nash Equilibria composition). *Let $A = \langle N, \mathcal{S}_A, u_A \rangle$ and $B = \langle N, \mathcal{S}_B, u_B \rangle$ be two games in normal form representation. Then, $\{(\sigma_{Ai}, \sigma_{Bi})\}$ is a Nash Equilibrium for $A \odot B$ if and only if $\{\sigma_{Ai}\}$ and $\{\sigma_{Bi}\}$ are Nash Equilibria respectively for A and B.*

Moreover, the operator composition is not only closed with respect to Nash Equilibria, but also closed with respect to the property of practicality. Let $A = \langle N, \mathcal{S}_A, u_A \rangle$ and $B = \langle N, \mathcal{S}_B, u_B \rangle$ be two games and let (A, σ_A) and (B, σ_B) be two practical mechanisms. Then, $(A \odot B, \{\sigma_{Ai}, \sigma_{Bi}\})$ is a practical mechanism.

Concerning robustness properties for composition of games, we can state the following results on resiliency and weak immunity for two composed games. The results can be generalized for the composition of multiple games.

▶ **Theorem 5** (resiliency). *Let* $A = \langle N, \mathcal{S}_A, u_A \rangle$ *and* $B = \langle N, \mathcal{S}_B, u_B \rangle$ *be two games and let* (A, σ_A) *and* (B, σ_B) *be two mechanisms respectively* k*-resilient and* k'*-resilient. Then,* $(A \odot B, \{\sigma_{Ai}, \sigma_{Bi}\})$ *is a* $\min(k, k')$*-resilient mechanism.*

▶ **Theorem 6** (weak immunity). *Let* $A = \langle N, \mathcal{S}_A, u_A \rangle$ *and* $B = \langle N, \mathcal{S}_B, u_B \rangle$ *be two games and let* (A, σ_A) *and* (B, σ_B) *be two mechanisms respectively* t*-weak-immune and* t'*-weak-immune. Then,* $(A \odot B, \{\sigma_{Ai}, \sigma_{Bi}\})$ *is a* $\min(t, t')$*-weak-immune mechanism.*

The first result states that given two k, k'-resilient mechanisms, the threshold on the maximum number of rational players allowed in the composition of games is the minimum among the number of rational players in the individual mechanisms. According to the second theorem, given two t, t'-weak immune mechanisms, the threshold on the maximum number of Byzantine players allowed in the composition of games is the minimum among the number of Byzantine players in the individual, as well. The proofs make use of the definition of k-resilience and t-weak immunity respectively; if a mechanism is k-resilient, then the protocol is followed if there are at most k rational players while if a mechanism is t-weak immune it provides non-negative outcomes if there are at most t Byzantine players.

3 Applications

In this section we prove the effectiveness of our framework by analyzing the robustness of different blockchain protocols. Section 3.1 and 3.2 analyze layer-1 protocols (Tendermint [22] and Bitcoin [25]) while Section 3.3 and 3.4 address layer-2 protocols (Lightning Network [30], a protocol on top of the Bitcoin blockchain and the side-chain protocol Platypus [29]). Finally, Section 3.5 analyzes a cross-chain swap protocol [27] allowing two users to exchange cryptoassets belonging to two different blockchains. Names of the variables in the following sections are consistent with the notation used in the papers where protocols are introduced.

3.1 Tendermint

This section addresses the Tendermint consensus (i.e., Tendermint-core [22, 4]) which is characterized by three rounds: the Pre-Propose round, the Propose round and the Vote round. During the Pre-Propose round, the proposer presents a block to the other participants. During the Propose round, each participant chooses whether to accept or not the block and broadcasts her decision. If the votes for the proposal exceed a predetermined threshold ν then participants start the Vote phase. If the block receives more than ν votes, it is validated. Tendermint's consensus algorithm sets $\nu = n - f = \frac{2}{3}n$; the threshold representing the number of non-faulty actors (as n denotes the total number of nodes and f the total number of faulty nodes) is set to $\frac{2}{3}$ of the network participants.

▶ **Definition 7.** *The Tendermint game is a mechanism* $(\Gamma^{tc}, \sigma^{tc})$ *such that the game* Γ^{tc} *represents the decision-making problem and the strategy* σ^{tc} *is the prescribed consensus protocol. Once a proposal* v *is received,* N *players choose either to check or not to check the validity of the value, then they can choose either to Vote or Not to Vote for it. At the very first stage of the game (stage a) a player can choose either to check (*C*) the validity or not check (*NC*). If she checks it, she can choose to Vote or Not Vote for it, in case value* v *is valid (stage b) or not (stage c). If she does not check it (stage d), she can choose to*

Vote (V) or Not Vote (NV) for it. Every strategy τ is represented by a vector (a, b, c, d) in which $a \in \{C, NC\}$, $b, c, d \in \{V, NV\}$. The utility for player i is $u_i(\tau) = 1$ if a valid block is approved or a non-valid block is not approved, $u_i(\tau) = 0$ if a valid block is not approved and $u_i(\tau) < 0$ if a non-valid block is approved.

The strategy prescribed by Tendermint's consensus protocol is $\sigma^{tc} = (C, V, NV, NV)$ i.e., to check for the validity of the proposal and then if the block is valid to vote for it, otherwise not vote for it. If the number of rational or Byzantine players allowed is $f < \frac{1}{3}n$, the other players have the necessary threshold to validate a block. Indeed, they can veto any validation of blocks proposed by malicious nodes. The mechanism $(\Gamma^{tc}, \sigma^{tc})$ is thus not f-weak-immune for any $f \geq \frac{1}{3}n$ and we can state the following results.

▶ **Theorem 8.** *The mechanism $(\Gamma^{tc}, \sigma^{tc})$ is (f, f)-robust for any $f < \frac{1}{3}n$.*

3.2 Bitcoin

Bitcoin is a permissionless blockchain based on the Proof-of-Work mechanism [25] where every user has a chance to publish a new block in the distributed ledger. The user probability to mine a new block is proportional to her computational power α. Bitcoin's protocol requires that once a block is mined, it should be broadcast to every other user. In case two or more blocks are mined at the same moment, the players split their effort to mine from any of the blocks (i.e., a *fork* is generated). Hence, published blocks are not automatically validated; they are considered as valid when belonging to the *longest chain* i.e., the longest branch of the ledger called *main chain.* A valid block generates a reward to the users who mined it.

As for Tendermint, Bitcoin's protocol can be represented by a mechanism $(\Gamma^{btc}, \sigma^{btc})$. We take into account the worst-case scenario, in which the Byzantine users coordinate, thus they are represented by a single player i. The altruistic users act in the same way and can therefore be represented by a second player j. The strategies of the players correspond to choosing (i) where in the chain add a new block and (ii) when to publish the mined blocks. Player j plays only one strategy defined by σ^{btc} i.e., she follows the protocol by mining on the main chain or splitting her effort if there is more than one chain of the same length available. Since the game is stochastic, we group all the equivalent states of the game in the same class; we consider two states as equivalent if they have the same configuration independently from the precise position in the chain (i.e., the difference between the number of mined blocks by the i and j is the same). In the Bitcoin blockchain a best practice is to consider a block as valid if belonging to a chain where at least B (usually, $B = 6$) blocks have been published afterwards, because it is presumably considered impossible to create a longer chain that does not include it. This block is invalidated if a fork is made at the previous block and more than $B + 1$ blocks are published starting from it. In this way, the block does not belong to the longest chain anymore and it is not considered as valid.

▶ **Definition 9.** *The Bitcoin game is a mechanism $(\Gamma^{btc}, \sigma^{btc})$ such that the game Γ^{btc} represents the decision-making problem and the strategy σ^{btc} is the prescribed protocol. The game Γ^{btc} is characterized by two players i and j, who have respectively mining power α and $1 - \alpha$ and every state of the game can be represented by the state class $\{x_k\}_{k \in \{0,1,\ldots,B+1\}}$, where x_k is the number of blocks mined, yet not published, at level k by player i. The block at level $k = 0$ is the only one to be published. The initial state of the game is $\{x_k = 0\}$ $\forall k \in \{0, 1, \ldots, B + 1\}$, while the final state of the game is represented by the state class with value $x_{B+1} \geq 1$. While player j has only one possible strategy σ^{btc}, player i can choose which branches to mine from (i.e. at which level k add the block). The utility of the players is the number of bitcoins they own according to the published blocks on the longest chain.*

The game theoretical framework let us state the following results on Bitcoin's mechanism robustness. Any subset of players T with $|T| = t$ having mining power $\alpha > 0$ have a small probability, not negligible, to perform a successful attack, by building a longer chain which does not include a block which was already considered valid (Theorem 10).

▶ **Theorem 10.** *The Bitcoin mechanism* $(\Gamma^{btc}, \sigma^{btc})$ *is not* t*-weak-immune for any* t*.*

▶ **Theorem 11.** *The Bitcoin mechanism* $(\Gamma^{btc}, \sigma^{btc})$ *is* k*-resilient if* k *players have at most* $\alpha \leq \frac{3}{20}$ *as total mining power.*

On the long run the majority of users ($\alpha \geq \frac{1}{2}$) produce the longer chain. However, on the short run a minority of users ($\alpha < \frac{1}{2}$) can make a fork on the longer chain with positive probability. The following theorem provides the value of this probability.

▶ **Theorem 12.** *The probability for a Byzantine player with computation power* α*, with* $\alpha < \frac{1}{2}$*, to prevent a transaction to be published within* $\Delta > 0$ *blocks is:*

$$\Phi_\Delta(\alpha) = \frac{\alpha}{1-\alpha} - \sum_{k=1}^{\Delta-1}(1 - \Phi_{\Delta-k}(\alpha)) \cdot \alpha^k \cdot (1-\alpha)^k \cdot M(k) \quad \textit{where,}$$

$M(k)$ *is a function defined in [17] mapping natural numbers to the sequence* $1, 1, 2, 5, 13, 42 \ldots$.

3.3 Lightning Network

In the Bitcoin blockchain transactions are collected in blocks, validated and published on the ledger. Bitcoin faces a problem of scalability, in terms of speed, volume and value of the transactions. In order to overcome these issues authors in [30] introduce a layer-2 class of protocols called Lightning Network. The latter allows users to create bidirectional payment *channels* to handle unlimited transactions in a private manner i.e., off-chain without involving the blockchain. Two users A and B open a channel by publishing on the Bitcoin blockchain two transactions towards a fund F. The amounts of the two transactions constitute the initial balance of the channel. In Section 3.3.1 we analyze the module to open a channel. The fund F can send or receive cryptoassets via blockchain transactions only if both users sign them. Once the channel is opened, users can exchange by simply privately updating the balance of the channel (cf. Section 3.3.2). The protocol to update the balance is discussed in Section 3.3.3. A further construction allowing users to create transactions within the channel that can be triggered at will is adopted in the protocol to update the balance (cf. Section 3.3.4). When the users decide to close the channel, two transactions are published on the Bitcoin blockchain: one from F to A and another from F to B. The value of the transactions corresponds to the ones of the latest balance. The protocol to close the channel is presented in Section 3.3.2. Lightning Network allows transactions also between users who have not opened a common channel (i.e., *routed payment*). Indeed, two users can perform a transaction through a path of open channels, using other users as intermediate nodes. This protocol is analyzed in Section 3.3.4.

3.3.1 Opening module

In order to open a channel, the Bitcoin users create a transaction Tx towards F and two different commitments ($C1a$ for A and $C1b$ for B) letting them close the channel unilaterally. The protocol [30] specifies in which order the commitments Tx, $C1a$ and $C1b$ have to be signed by the users. We formalize the protocol with a game in extensive form Γ^{op} (cf.

Definition 13) where at every node of the tree (i.e., decision step) the player involved in the protocol has two actions available: either following it by signing the commitment required or not following it. The *initial state* corresponds to having no channel opened, while the final state corresponds to having the channel opened. We assign "null" utility to the initial state and positive utility (by convention fixed to 1) to the final state. If at any step the players do not follow the protocol, they get back to the initial state with outcome $(0,0)$. If they do follow it at every step, they are able to open the channel having as an outcome $(1,1)$. We denote by $\sigma^{op} = (\{C1b_{A\cdot}, Tx_{A\cdot}\}, \{C1a_{\cdot B}, Tx_{AB}\})$ the strategy profile recommended by the protocol in which the actions are played respectively at nodes $(\{1,3\},\{2,4\})$.

▶ **Definition 13.** *The opening game Γ^{op} is a game in extensive form, with two players $\{A,B\}$ and 4 nodes, labeled by a number (1 is the root):*

1. *A has two actions available: $C1b_{\cdot\cdot}$ provides outcome $(0,0)$; $C1b_{A\cdot}$ leads to node 2.*
2. *B has two actions available: $C1a_{\cdot\cdot}$ provides outcome $(0,0)$; $C1a_{\cdot B}$ leads to node 3.*
3. *A has two actions available: $Tx_{\cdot\cdot}$ provides outcome $(0,0)$; $Tx_{A\cdot}$ leads to node 4.*
4. *B has two actions available: $Tx_{A\cdot}$ provides outcome $(0,0)$; Tx_{AB} provides outcome $(1,1)$.*

The protocol is thus represented by the mechanism $(\Gamma^{op}, \sigma^{op})$, whose properties we analyze in the sequel.

▶ **Theorem 14.** *The mechanism $(\Gamma^{op}, \sigma^{op})$ is not immune.*

The mechanism would be immune if both players receive no lower payoff than $u(\sigma^{op}) = (1,1)$, no matter what the other player chooses. A counterexample is B deviating from $\sigma^{op}_B = \{C1a_{\cdot B}, Tx_{AB}\}$ to $\tau_B = \{C1a_{\cdot\cdot}, Tx_{AB}\}$, i.e. B refusing to signing $C1a$ at step 2. For player A the outcome of $u_A(\sigma^{op}_A, \tau_B) = 0 < 1 = u(\sigma^{op})$.

▶ **Theorem 15.** *The mechanism $(\Gamma^{op}, \sigma^{op})$ is optimal resilient and weak immune.*

3.3.2 Classical and alternative closing modules

As described in Section 3.3.1 both users A and B can unilaterally close the channel by publishing respectively on the blockchain commitment $C1a$ and $C1b$. If a user decides to unilaterally close the channel, she receives her part of the fund after that a given number Δ of blocks are validated on the Bitcoin blockchain, while the other user receives it immediately. The protocol recommends to close the channel by creating a new transaction, namely ES, that let the players receive their cryptoasset immediately. We model the problem with the following game in normal form.

▶ **Definition 16.** *The closing game $\Gamma^{cl} = \langle N, \mathcal{S}, u\rangle$ of the channel with balance (x_A, x_B) with $x_A, x_B > 0$ is a game in normal form, with two players $\{A,B\}$ who have available three different pure strategies each: $\mathcal{S}_A = \{C1a_{AB}, DN, ES\}$ and $\mathcal{S}_B = \{C1b_{AB}, DN, ES\}$. The value of the utility can be found in the following payoff table.*

			B	
		$C1b_{AB}$	DN	ES
	$C1a_{AB}$	$(\frac{1}{2},\frac{1}{2})$	$(0,1)$	$(0,1)$
A	DN	$(1,0)$	$(-1,-1)$	$(-1,-1)$
	ES	$(1,0)$	$(-1,-1)$	$(1,1)$

First, we assume that the channel (x_A, x_B) is funded by both players i.e., $x_A, x_B > 0$. If one of the two players has no asset involved in the channel, we have to model the problem with a degenerate game, in which she can arbitrarily play any possible strategy. We recommend users to never unilaterally fund the channel. The players have three different strategies: publishing their commitment, seeking a deal to create a new transaction ES or just doing nothing DN. We assign null utility to players who receive their asset after Δ blocks, positive utility (normalized to 1) if they receive it immediately, negative utility if they cannot redeem their cryptoassets. If they both try and publish their commitment $(C1a_{AB}, C1b_{AB})$ we assume they have equal probability to get their commitment published first. The protocol recommends the strategy profile $\sigma^{cl} = (ES, ES)$ i.e., both players seek a deal. In the following we analyze the properties of the mechanism $(\Gamma^{cl}, \sigma^{cl})$.

▶ **Theorem 17.** *Under the assumption $x_A > 0$ or $x_B > 0$, the mechanism $(\Gamma^{cl}, \sigma^{cl})$ is optimal resilient, but not weak immune.*

To prove that the mechanism is not weak immune it is sufficient to show a counterexample. Indeed, if A chooses ES as required by the protocol and B chooses the Byzantine strategy N, player A receives a negative outcome $u_A(\sigma^{cl}_A, DN) = u_A(ES, DN) = -1$. Since the mechanism is not weak immune, it is not immune either. We thus provide an alternative protocol that satisfies the property of weak immunity.

▶ **Theorem 18.** *Under the assumption $x_A > 0$ or $x_B > 0$, the only weak immune mechanism is (Γ^{cl}, σ^*) with $\sigma^* = (C1a_{AB}, C1b_{AB})$.*

If users play this strategy, they never get a negative utility if the other player deviates. It is easy to prove that this is the only strategy profile with this property.

3.3.3 Updating module

Performing a transaction within a channel consists in updating its balance. Technically, the previous commitments ($C1a$ and $C1b$) with balance (x_A, x_B) are replaced by two new commitments ($C2a$ and $C2b$) with different balance (x'_A, x'_B). In order to prevent players from publishing old commitments, they sign two Breach Remedy Transactions ($BR1a$ and $BR1b$), that can invalidate $C1a$ and $C2b$. Indeed, if any party publishes an outdated commitment the other one can retrieve all the cryptoassets in the fund. If, for instance, A publishes the outdated commitment $C1a$, she can retrieve her fund x_A unless B publishes $BR1a$ before Δ blocks are validated. The protocol to update the balance requires the players to sign the commitments in a specific order [30]. We formalize the protocol with a game in extensive form Γ^{up} (cf. Definition 19). The initial state corresponds to the previous balance (with null utility), the final state to the updated balance (with utility equal to 1). We assign a negative value to the states in which players lose their cryptoassets or part of them.

▶ **Definition 19.** *The updating game Γ^{up} is a game in extensive form, with two players $\{A, B\}$ and 5 nodes, labeled by a number (1 is the root):*

1. *A plays. $C2b_{..}$ provides outcome $(0,0)$; $C2b_{A.}$ leads to node 2.*
2. *B plays. $C2a_{..}$ provides outcome $(0,0)$; $C2b_{AB}$ provides outcome $(1,1)$; $C2a_{.B}$ leads to node 3.*
3. *A plays. $BR1a_{..}$ provides outcome $(0,0)$; $C2a_{AB}$ provides outcome $(1,1)$; $BR1a_{A.}$ leads to node 4.*
4. *B plays. $BR1b_{.B}$ provides outcome $(1,1)$; $BR1b_{..}$ leads to node 5.*
5. *A plays. $C1a_{AB}$ provides outcome $(-1,1)$; $C2a_{AB}$ provides outcome $(1,1)$.*

The protocol recommends to sign all the commitments and it is thus represented by the strategy profile $\sigma^{up} = (\{C2b_{A.}, BR1a_{A.}, C2a_{AB}\}, \{C2a_{.B}, BR1b_{.B}\})$. We analyze the mechanism $(\Gamma^{up}, \sigma^{up})$ under the assumption that it is always possible to publish a transaction within Δ blocks, otherwise it is not possible to validate the breach remedy transactions in time. The probability that this happens when a Byzantine agent with computational power α attacks the Bitcoin blockchain is $1 - \Phi_\Delta(\alpha)$ (cf. Theorem 12).

▶ **Theorem 20.** *The mechanism* $(\Gamma^{up}, \sigma^{up})$ *is optimal resilient and weak immune with probability* $1 - \Phi_\Delta(\alpha)$, *but it is not immune.*

3.3.4 Routing module

Lightning Network provides a protocol, called *Hash time Locked Contract* (HTLC), that allows to create transactions that can be triggered at will. The protocol for the HTLC works as follows: (i) user A creates a pair (H, R), where H is public and R is its private key; (ii) she shares with user B a commitment together with the string H; (iii) once this commitment is published on the Bitcoin blockchain, user B can receive the transaction only if she can provide the private key R within Δ blocks. It is easy to check that R is the private key of H, but it is almost impossible to retrieve R, given H. In this way, user A can trigger the transaction whenever she wants by disclosing R to user B. The protocol can be represented by a mechanism $(\Gamma^{htlc}, \sigma^{htlc})$, that has the very same structure of the updating module (cf. Section 3.3.3) and thus satisfies optimal resilience and weak immunity, but not immunity.

The HTLC is implicated in the protocol allowing users to perform transactions also if they do not share a common channel. Indeed, it is sufficient that among the two users there is a path of channels i.e., a sequence of users who two-by-two share a channel. For instance, let us suppose that users A and C have both opened a separate channel with a third user B. In the *routed payment* user B is the intermediate node. The model can be easily generalised to any number of intermediate nodes. Routing fees are not included, but they would not change the solution of the game. The protocol for routed payment works as follows: (i) user C creates a pair of strings (H, R) and then discloses H to user A; (ii) user A creates an HTLC with user B locked with the public key H then, (iii) user B creates an HTLC with user C locked with H; (iv) finally, user C discloses R with user B and triggers the transaction, and so does user B with user A. In this way, user C receives the payment, user A sends it and user B gains from a channel with A what she loses from the channel with C. In practice, the value of the two transactions do not coincide, so that the difference consists in the fee to be provided to user B. We formalize the protocol with a game in extensive form Γ^{rout}. The strategy profile recommended by the protocol is denoted by $\sigma^{rout} = (\{H_A^{AB}\}, \{H_B^{BC}, Y\}, \{Y, Y\})$.

▶ **Definition 21.** *The routing game* Γ^{rout} *is a game in extensive form, with three players* $\{A, B, C\}$ *and 5 nodes, labeled by a number (*1 *is the root):*

1. C *has two actions available: either* N, *not sending* H *to* A, *which provides outcome* $(0, 0, 0)$, *or* Y, *sending* H *to* A, *which leads to node* 2.
2. A *has two actions available: either* $H_{\cdot}^{AB}$, *which provides outcome* $(0, 0, 0)$, *or* H_A^{AB}, *which leads to node* 3.
3. B *has two actions available: either* $H_{\cdot}^{BC}$, *which provides outcome* $(0, 0, 0)$, *or* H_B^{BC}, *which leads to node* 4.
4. C *has two actions available: either* N, *not disclosing* R *to* B, *which provides outcome* $(0, 0, 0)$, *or* Y, *disclosing* R *to* B, *which leads to node* 5.
5. B *has two actions available: either* N, *not disclosing* R *to* A, *which provides outcome* $(1, -1, 1)$ *or* Y, *disclosing* R *to* A, *which provides outcome* $(1, 1, 1)$.

▶ **Theorem 22.** *Under the assumption that in both HTLCs the transactions can be triggered,* $(\Gamma^{rout}, \sigma^{rout})$ *is optimal resilient and weak immune, but it is not immune.*

The HTLCs introduced in the protocol work independently from the routing protocol. We can model them with two different mechanisms: $(\Gamma^{AB}, \sigma^{AB})$ for H^{AB} and $(\Gamma^{BC}, \sigma^{BC})$ for H^{BC}. The mechanism $(\Gamma^{AB}, \sigma^{AB})$ represents the HTLC deployed on the channel A-B, while the mechanism $(\Gamma^{BC}, \sigma^{BC})$ refers to the HTLC implemented on the channel B-C. The HTLCs belong to two different channels, so they are independent one from another. The assumption from the routing protocol is that in both HTLCs the transactions can be triggered, but this is true only if every transaction can be published within Δ blocks. Under this assumption, the routed payment is represented by three independent protocols $(\Gamma^{rout}, \sigma^{rout})$, $(\Gamma^{AB}, \sigma^{AB})$, and $(\Gamma^{BC}, \sigma^{BC})$. Therefore, we analyze the properties of its mechanism by defining and analyzing the composition of the three games $(\Gamma^{rout} \odot \Gamma^{AB} \odot \Gamma^{BC}, \{\sigma_i^{rout}, \sigma_i^{AB}, \sigma_i^{BC}\})$.

▶ **Theorem 23.** *The mechanism $(\Gamma^{rout} \odot \Gamma^{AB} \odot \Gamma^{BC}, \{\sigma_i^{rout}, \sigma_i^{AB}, \sigma_i^{BC}\})$ is optimal resilient and weak immune with probability $1 - \Phi_\Delta(\alpha)$ (cf. Theorem 12).*

Proof. The operator composition (cf. Definition 3) is invariant with respect the properties of the mechanisms. Thanks to Theorems 20, 22 we have that $(\Gamma^{rout}, \sigma^{rout})$, $(\Gamma^{AB}, \sigma^{AB})$ and $(\Gamma^{BC}, \sigma^{BC})$ are practical hence their composition $(\Gamma^{rout} \odot \Gamma^{AB} \odot \Gamma^{BC}, \{\sigma_i^{rout}, \sigma_i^{AB}, \sigma_i^{BC}\})$ is practical. Analogously, thanks to Theorems 20, 22 we have that every single mechanism is k-resilient for all k and t-weak-immune for all t. Theorems 5, 6 allow us to say that the composition $(\Gamma^{rout} \odot \Gamma^{AB} \odot \Gamma^{BC}, \{\sigma_i^{rout}, \sigma_i^{AB}, \sigma_i^{BC}\})$ is k-resilient for all k and t-weak-immune for all t, i.e., it is strongly resilient and weak immune. ◀

Recap. All the results of the Lightning Network are available in Table 1. The Lightning Network is built on top of Bitcoin blockchain therefore its properties depend highly on Bitcoin blockchain's ones. If we exclude the closing protocol, the Lightning Network satisfies optimal resilience and weak immunity. Hence, we can compose (cf. Definition 3) Lightning Network protocols' games with Bitcoin mechanism's (that provide weaker results, cf. Section 3.2) and prove that the Lightning Network satisfies the same properties of the Bitcoin mechanism.

3.4 Side-chain

A different solution to overcome the scalability and privacy problems of permissionless blockchains is offered by Platypus [29], a protocol that allows a group of users to create a childchain (sidechain) that can handle off-chain transactions without the need of synchrony among peers. This section analyzes the protocol to create a Platypus chain proposed in [29]. In this section we would like to extend the analysis performed in [29] proving new properties which fit our framework. The protocol lets the childchain validators broadcast transactions to the peers until the number of validators who have confirmed the transactions overcome a defined threshold. The protocol is divided into phases consisting of players acting at the same time, indeed it is possible to model this protocol with a game in extensive form Γ^{cr}, in which players are split into two categories: normal users (set U) and the validators (set V). Users' utility is positive if their transactions are successfully published and it is negative if different transactions are validated instead.

▶ **Definition 24.** *The creation game is a game Γ^{cr} in extensive form, where $U \cup V$ is the set of players, with $m_v = |U \cup V|$. Every phase corresponds to a node of the tree, at which players play at the same time.*

- ***Phase 1****; only the player p_0 is involved. The player p_0 has two actions: either complete the transaction Y or not N. If she does not, the outcome is 0 for all players.*

- ***Phase 2;*** *every player within normal users play at the same time. Everyone has available the same two actions: broadcasting their transaction Y or not N. If the transaction is not broadcast for player i, her utility is always 0.*
- ***Phase 3;*** *the validators can choose within a set of actions a_u with $u \subseteq U$ i.e., they can validate all the transaction for the users within the set u. The cardinality of the set of their actions is equal to $2^{|U|}$. The utility for the validators corresponds to the number of valid transactions which are broadcast.*
- ***Phase 4;*** *the validators can choose within a set of actions in the form $(b_t, s_{t'})$, where t and t' are any subset of transactions broadcast in Phase 3. The action b_t consists in broadcasting the transactions belonging to the set t until $\lfloor 2m_v/3 \rfloor + 1$ validators receive it, while $s_{t'}$ means to send the transactions in t'.*

We define the mechanism $(\Gamma^{cr}, \sigma^{cr})$, where $\sigma^{cr} \in \mathcal{S}$ is the strategy of following the protocol i.e., for normal users u the strategy is $\sigma_u^{cr} = Y$, while for validators v the strategy is $\sigma_v^{cr} = (a_{u^*}, b_{t^*}, s_{t^*})$, where u^* is the set of users who send a message and t^* is the set of transactions broadcast in Phase 3. We thus analyze the properties of the mechanism.

▶ **Theorem 25.** *The mechanism $(\Gamma^{cr}, \sigma^{cr})$ is optimal resilient and $\lfloor \frac{m_v}{3} \rfloor$-weak-immune, but not t-immune for any t.*

In [29] it is proved that no wrong transaction can be validated if there are at most $\lfloor \frac{m_v}{3} \rfloor$ corrupted players. This property cannot be expressed with the concept of immunity, which is too strong. Hence, to capture this information we exploit the definition of t-weak-immunity (cf. Definition 1). Within our model, the upper bound on the number of corrupted players means that no negative payoff is given to the players under the hypothesis that there are at most $\lfloor \frac{m_v}{3} \rfloor$ Byzantine nodes i.e., that the mechanism is $\lfloor \frac{m_v}{3} \rfloor$-weak-immune.

3.5 Cross-chain swap

In this section we analyze the protocol introduced in [27] allowing two users to swap assets that belong to two different blockchains which do not communicate with each other. In [15] the authors introduce a theoretical framework proving that the protocol is correct for those players who are altruistic, no matter what the others do. In the following we prove that the Cross-chain swap protocol [27] satisfies the (k, t)-weak-robustness. In this protocol users publish two different transactions on two different blockchains (e.g., Altcoin and Bitcoin) that can be triggered with the disclosure of a single private key x by means of hashed time lock contracts (HTLCs, cf. Section 3.3). The transactions have to be published within two different time intervals, Δ_1 and Δ_2 (where $\Delta_1 \geq 2\Delta_2$), depending on the corresponding blockchain. In a 2-player context authors in [27, 15] assume that the transactions can be published within the time interval $[0, \min(\Delta_1, \Delta_2)] = [0, \Delta_2]$. Since the two blockchains are independent we model the protocol with two different mechanisms $(\mathcal{G}_1, \sigma_1)$ and $(\mathcal{G}_2, \sigma_2)$ (cf. Definitions 26 and 27), representing the actions that players perform in each blockchain (i.e., $(\mathcal{G}_1, \sigma_1)$ for the Bitcoin blockchain and $(\mathcal{G}_2, \sigma_2)$ for the Altcoin blockchain).

▶ **Definition 26.** *The Bitcoin game is an extensive form game $\mathcal{G}_1$ with 2 players $\{A, B\}$ and 5 nodes (1 is the root):*

1. *A can either (Y) create TX1 and TX2, that leads to node 2 or (N) not create them, with outcome $(0, 0)$.*
2. *B can either (Y) sign TX2, that leads to node 3, or (N) refuse to do it, with outcome $(0, 0)$.*

3. *A can either (N) do nothing, with thus outcome $(0,0)$, or (Y) publish TX1 on the Bitcoin blockchain, that leads to node 4.*
4. *Both A and B have available two actions: either (Y) publish TX2 before secret x is revealed or (N) not publish it. If any of the two users does so, the outcome is $(0,0)$. Otherwise, A reveals secret x and (N,N) leads to node 5.*
5. *B can either (Y) publish secret x on the Bitcoin blockchain or (N) not publish it. If she does, the outcome is $(1,1)$. If she does not, the outcome is $(1,-1)$.*

The strategy profile that corresponds to following the protocol is $\sigma_1 = (\{Y,Y,N\},\{Y,N,Y\})$.

▶ **Definition 27.** *The Altcoin game is an extensive form game $\mathcal{G}_2$ with 2 players $\{A,B\}$ and 5 nodes (1 is the root):*

1. *B can either (Y) create TX3 and TX4, or (N) do nothing. The action Y leads to node 2, while the action N leads to the outcome $(0,0)$.*
2. *A can either (Y) sign TX4, that leads to node 3, or (N) refuse to do it, with outcome $(0,0)$.*
3. *B can either (N) do nothing, with thus outcome $(0,0)$, or (Y) publish TX3 on the Altcoin blockchain, that leads to node 4.*
4. *Both A and B have available two actions: either (Y) publish TX4 before secret x is revealed or (N) not publish it. If any of the two does so, the outcome is $(0,0)$. Otherwise, A reveals secret x and (N,N) leads to node 5.*
5. *A can either (Y) publish secret x on the Altcoin blockhain or (N) not publish it. If she does, the outcome is $(1,0)$. If she does not, the outcome is $(0,0)$.*

The strategy profile that corresponds to following the protocol is $\sigma_2 = (\{Y,N,Y\},\{Y,Y,N\})$.

Since the two blockchains are independent, we consider the composition of the two games $(\mathcal{G}_1 \odot \mathcal{G}_2, \{\sigma_{1i}, \sigma_{2i}\})$ representing the full protocol and analyze it.We can easily see that the mechanism is not immune, indeed it is sufficient that one player does not create or publish a transaction to stop the protocol. However, we have the following important result.

▶ **Theorem 28.** *Under the assumption that any transaction can be published within a time interval $[0,\Delta_2]$, the mechanism $(\mathcal{G}_1 \odot \mathcal{G}_2, \{\sigma_{1i}, \sigma_{2i}\})$ is optimal resilient and weak immune, but it is not immune.*

4 Conclusions

We proposed the first generic game theoretical framework that models the robustness of blockchains towards rational and Byzantine behaviors. In this paper we identified the necessary and sufficient conditions for a protocol to be robust (defined as the conjunction of two properties: k-resilience and t-weak immunity) and developed a methodology to characterize the robustness of complex protocols via the composition of simpler robust building blocks. The effectiveness of our framework was demonstrated by its capability to capture the robustness of various blockchain protocols such as Bitcoin, Tendermint, lightning networks (original and alternative closing modules), side-chain and cross-chain protocols. Our work continues the work of [1] that introduced the notion of robustness defined in terms of t-immunity and k-resilience. The framework of [1] was never used till our study in the context of blockchain protocols. Using the framework of [1] we proved that a large class of blockchain protocols (cf. Table 1) does not satisfy the t-immunity property. It should be noted that our negative result related to the t-immunity property does not depend on the specific choice of a utility function. Therefore, we proposed a relaxation of this property i.e., t-weak immunity. We analysed the k-resilience and the t-weak immunity of a large class of blockchain protocols, providing bounds on respectively the number of rational and Byzantine processes (cf. results in Table 1).

These results are based on strict hypotheses under which the model we introduced takes into account all the possible alternatives to the protocol. As future work we plan to relax these hypotheses and provide more accurate estimation of the robustness indices. Moreover, we plan to investigate the resilience of other blockchain protocols such as Algorand [10] or DAG-based blockchains (e.g., Spectre [33], Phantom [34] or IOTA [31]). A further possible direction of research is an extension of our framework in order to analyse repeated consensus protocols (e.g., protocols presented in [4]).

References

1 Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: Robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 53–62, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1146381.1146393.

2 Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *SOSP '05*, 2005.

3 Yackolley Amoussou-Guenou, Bruno Biais, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Rational vs byzantine players in consensus-based blockchains. In Amal El Fallah Seghrouchni, Gita Sukthankar, Bo An, and Neil Yorke-Smith, editors, *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, pages 43–51. International Foundation for Autonomous Agents and Multiagent Systems, 2020. URL: https://dl.acm.org/doi/abs/10.5555/3398761.3398772.

4 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting tendermint. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Networked Systems*, pages 166–182, Cham, 2019. Springer International Publishing.

5 Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. Brick: Asynchronous incentive-compatible payment channels. In *International Conference on Financial Cryptography and Data Security*, 2021.

6 Marianna Belotti, Stefano Moretti, Maria Potop-Butucaru, and Stefano Secci. Game theoretical analysis of Atomic Cross-Chain Swaps. In *40th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Singapore, Singapore, 2020. URL: https://hal.archives-ouvertes.fr/hal-02414356.

7 Marianna Belotti, Stefano Moretti, and Paolo Zappalà. Rewarding miners: bankruptcy situations and pooling strategies. In *17th European Conference on Multi-Agent Systems (EUMAS)*, Tessaloniki, Greece, 2020. URL: https://hal.archives-ouvertes.fr/hal-02481155.

8 Iddo Bentov, Pavel Hubácek, Tal Moran, and Asaf Nadler. Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies. *IACR Cryptology ePrint Archive*, 2017:300, 2017.

9 B.Douglas Bernheim, Bezalel Peleg, and Michael D Whinston. Coalition-proof nash equilibria i. concepts. *Journal of Economic Theory*, 42(1):1–12, 1987. doi:10.1016/0022-0531(87)90099-8.

10 Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019. doi:10.1016/j.tcs.2019.02.001.

11 Altannar Chinchuluun, Panos Pardalos, Athanasios Migdalas, and Leonidas Pitsoulis. *Pareto Optimality, Game Theory And Equilibria*, volume 17. Springer, 2008. doi:10.1007/978-0-387-77247-9.

12 Christian Ewerhart. Finite blockchain games. *Economics Letters*, 197:109614, 2020. doi:10.1016/j.econlet.2020.109614.

13 Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

14 Peter Hammond. *Utility Invariance in Non-Cooperative Games*, volume 38, pages 31–50. Springer, June 2006. doi:10.1007/0-387-25706-3.

15 Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

16 John Hillas. On the definition of the strategic stability of equilibria. *Econometrica*, 58(6):1365–1390, 1990. URL: http://www.jstor.org/stable/2938320.

17 OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2021. URL: https://oeis.org/A178682.

18 Aggelos Kiayias and Aikaterini-Panagiota Stouka. Coalition-safe equilibria with virtual payoffs. *arXiv preprint*, 2019. arXiv:2001.00047.

19 Elon Kohlberg and Jean-Francois Mertens. On the strategic stability of equilibria. *Econometrica: Journal of the Econometric Society*, pages 1003–1037, 1986.

20 Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In Christoph Meinel and Sophie Tison, editors, *STACS 99*, pages 404–413, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

21 Harold William Kuhn and Albert William Tucker. *Contributions to the Theory of Games*, volume 2. Princeton University Press, 1953.

22 Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

23 Z. Liu, N. C. Luong, W. Wang, D. Niyato, P. Wang, Y. Liang, and D. I. Kim. A survey on blockchain: A game theoretical perspective. *IEEE Access*, 7:47615–47643, 2019.

24 Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. When selfish meets evil: Byzantine players in a virus inoculation game. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, volume 2006, pages 35–44, January 2006. doi:10.1145/1146381.1146391.

25 Satoshi Nakamoto. A peer-to-peer electronic cash system, 2008.

26 John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950. doi:10.1073/pnas.36.1.48.

27 Tier Nolan. Re: Alt chains and atomic transfers. accessed on January 10, 2020. https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949.

28 Rafael Pass and Elaine Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 315–324, 2017.

29 Alejandro Ranchal Pedrosa and Vincent Gramoli. Platypus: Offchain protocol without synchrony. In Aris Gkoulalas-Divanis, Mirco Marchetti, and Dimiter R. Avresky, editors, *18th IEEE International Symposium on Network Computing and Applications, NCA 2019, Cambridge, MA, USA, September 26-28, 2019*, pages 1–8. IEEE, 2019. doi:10.1109/NCA.2019.8935037.

30 Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

31 Serguei Popov, Olivia Saa, and Paulo Finardi. Equilibria in the tangle. *Comput. Ind. Eng.*, 136:160–172, 2019. doi:10.1016/j.cie.2019.07.025.

32 Okke Schrijvers et al. Incentive compatibility of bitcoin mining pool reward functions. In *International Conference on Financial Cryptography and Data Security*, pages 477–498. Springer, 2016.

33 Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptol. ePrint Arch.*, 2016:1159, 2016.

34 Yonatan Sompolinsky and Aviv Zohar. Phantom. *IACR Cryptology ePrint Archive, Report 2018/104*, 2018.

35 Itay Tsabary and Ittay Eyal. The gap game. In *Proceedings of the 2018 ACM SIGSAC conference on Computer and Communications Security*, pages 713–728, 2018.

36 Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials*, 18(3):2084–2123, 2016.

37 Wenbo Wang, Dinh Thai Hoang, Zehui Xiong, Dusit Niyato, Ping Wang, Peizhao Hu, and Yonggang Wen. A survey on consensus mechanisms and mining management in blockchain networks. *arXiv preprint*, pages 1–33, 2018. `arXiv:1805.02707`.

38 Paolo Zappalà, Marianna Belotti, Maria Potop-Butucaru, and Stefano Secci. Game theoretical framework for analyzing blockchains robustness. Technical report, Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, 2020. URL: `https://hal.archives-ouvertes.fr/hal-02634752/document`.

Brief Announcement: Fast Graphical Population Protocols

Dan Alistarh
IST Austria, Klosterneuburg, Austria

Rati Gelashvili
Novi Research, Menlo Park, CA, USA

Joel Rybicki
IST Austria, Klosterneuburg, Austria

Abstract

Let G be a graph on n nodes. In the stochastic population protocol model, a collection of n indistinguishable, resource-limited nodes collectively solve tasks via pairwise interactions. In each interaction, two randomly chosen neighbors first read each other's states, and then update their local states. A rich line of research has established tight upper and lower bounds on the complexity of fundamental tasks, such as majority and leader election, in this model, when G is a *clique*. Specifically, in the clique, these tasks can be solved *fast*, i.e., in $n \operatorname{polylog} n$ pairwise interactions, with high probability, using at most $\operatorname{polylog} n$ states per node.

In this work, we consider the more general setting where G is an arbitrary graph, and present a technique for simulating protocols designed for fully-connected networks in any connected regular graph. Our main result is a simulation that is *efficient* on many interesting graph families: roughly, the simulation overhead is polylogarithmic in the number of nodes, and quadratic in the conductance of the graph. As an example, this implies that, in any regular graph with conductance φ, both leader election and exact majority can be solved in $\varphi^{-2} \cdot n \operatorname{polylog} n$ pairwise interactions, with high probability, using at most $\varphi^{-2} \cdot \operatorname{polylog} n$ states per node. This shows that there are fast and space-efficient population protocols for leader election and exact majority on graphs with good expansion properties.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases population protocols, leader election, majority

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.43

Related Version *Full Version*: https://arxiv.org/abs/2102.08808

Funding *Joel Rybicki*: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 840605.

1 Introduction

In distributed computing, *population protocols* [4] have become a popular model for investigating the collective computational power of large collections of communication-bounded agents with limited computational capabilities. This model consists of n identical agents, seen as finite state machines, and computation proceeds via pairwise interactions of the agents, which trigger local state transitions. The sequence of interactions is provided by a scheduler, which picks pairs of agents to interact. Upon every interaction, the selected agents observe each other's states, and then update their local states. The goal is to have the system reach a configuration satisfying a given predicate, while minimising the number of interactions (time complexity) and the number of states per node (space complexity).

Early work on population protocols focused on the computational power of the model under various interaction graphs [4, 5]. More recently, the focus has shifted to complexity, often in the form of trade-offs between time and space complexity, e.g. [3, 17, 1, 9, 18, 16, 2].

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 43; pp. 43:1–43:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This line of work almost exclusively focuses on the *uniform* stochastic scheduler, where each interaction pair is chosen uniformly at random *among all pairs* of agents in the population, and the time complexity of a protocol is measured by the number of interactions needed to solve a task. However, many natural systems exhibit spatial structure and this structure can significantly influence the system dynamics.

Indeed, there is a separation in terms of computational power for population protocols in the clique versus other interaction graphs: connected interaction graphs can simulate adversarial interactions on the clique graph by shuffling the states of the nodes [4] and population protocols on some interaction graphs can compute a strictly larger set of predicates than protocols on the clique; see e.g. [6] for a survey of computability results.

By comparison, surprisingly little is known about the *complexity* of basic tasks in general interaction graphs under the stochastic scheduler. So far, only a handful of protocols have been analysed on general graphs. Existing analyses tend to be complex, and specialised to specific algorithms on limited graph classes [15, 11, 8]. This is natural: given the intricate dependencies which arise due to the underlying graph structure, the design and analysis of protocols in the spatial setting is understood to be challenging.

We provide a general approach showing that standard problems in population protocols can be solved *efficiently* under *graphical* stochastic schedulers, by leveraging solutions designed for complete graphs.

First, we give a general framework for simulating a large class of *synchronous* protocols designed for *fully-connected networks*, in the graphical stochastic population protocol model. Thus, the user can design efficient (and simple to analyse) synchronous algorithms on a clique model, and transport the analysis automatically to the population protocol model on a large class of interaction graphs. For instance, on any d-regular graph with edge expansion $\beta > 0$, the resulting overhead in parallel time and state complexity is in the order of $(d/\beta)^2 \cdot \operatorname{polylog} n$. As concrete applications, we show that for any d-regular graph with edge expansion $\beta > 0$, there exist protocols for leader election and exact majority that stabilise both in expectation and with high probability in $(d/\beta)^2 \cdot \operatorname{polylog} n$ parallel time, using $(d/\beta)^2 \cdot \operatorname{polylog} n$ states.

Second, to complement the results following from the simulation, we also show that, on any graph G with diameter $\operatorname{diam}(G)$ and m edges, leader election can be solved both in expectation and with high probability in $O(\operatorname{diam}(G) \cdot mn^2 \log n)$ parallel time, using a constant-state protocol. This result provides the first running time analysis of the protocol of [7].

Our reduction framework combines several techniques from different areas, and can be distilled down to the following ingredients.

We start by defining a simple *synchronous, fully-connected* model of communication for the n nodes, called the *k-token shuffling model.* This is the model in which the algorithm should be designed and analysed, and is similar, and in some ways simpler, relative to the standard population model. Specifically, nodes proceed in *synchronous* rounds, in which every node v first generates k tokens based on its current state. Tokens are then shuffled uniformly at random among the nodes. At the end of a round, every node v updates its local state based on its current state, and the tokens it received in the round. This simple model is quite powerful, as it can simulate both *pairwise* and *one-way* interactions between all sets of agents, for well-chosen settings of the parameter k.

Our key technical result is that any algorithm specified in this round-synchronous k-token shuffling model can be *efficiently* simulated in the graphical population model. Although intuitive, formally proving this result, and in particular obtaining bounds on the efficiency of the simulation, is non-trivial. First, to show that simulating *a single round* of the k-token shuffling model can be done efficiently, we introduce new type of *card shuffling process* [12, 10, 19], which we call the k-stack interchange process, and analyse its mixing time by linking it to random walks on the symmetric group.

Table 1 Protocols for exact majority (EM) and leader election (LE) for different graph classes. The state complexity is the number of states used by the protocol. The parallel time column gives the expected parallel time (expected number of interaction steps divided by n) to stabilise. (*) In [15], the running time of the protocol is bounded by the initial discrepancy in the inputs and the spectral properties of the contact rate matrix; bounds in terms of n are only given for select graph classes (paths, cycles, stars, random graphs and cliques). No sublinear in n bounds on parallel time are given in [15]. Protocols marked with ($\star$) stabilise also in non-regular graphs in poly(n) time.

Graph class	Task	States	Parallel time	Note
cliques	EM	4	$O(n \log n)$	[15]
	EM	$O(\log n)$	$\Theta(\log n)$	[13]
	LE	2	$\Theta(n)$	[14]
	LE	$\Theta(\log \log n)$	$\Theta(\log n)$	[9]
connected	EM	4	$\text{poly}(n)$	[15, 8], (*)
	LE	6	$O(\text{diam}(G) \cdot mn^2 \log n)$	**new** analysis of [7]
d-regular	EM	$(d/\beta)^2 \cdot \text{polylog}\, n$	$(d/\beta)^2 \cdot \text{polylog}\, n$	**new**, ($\star$)
	LE	$(d/\beta)^2 \cdot \text{polylog}\, n$	$(d/\beta)^2 \cdot \text{polylog}\, n$	**new**, ($\star$)

Second, to allow correct and efficient asynchronous simulation of the synchronous token shuffling model, we introduce two new gadgets: (1) a *graphical* version of *decentralised phase clocks* [1, 17], combined with (2) an *asynchronous* token shuffling protocol, which simulates the k-token interchange process in a graphical population protocol. The latter ingredient is our main technical result, as it requires both efficiently combining the above components, and carefully bounding the probability bias induced by simulating a synchronous model under asynchronous pairwise-random interactions.

Finally, we instantiate this framework to solve exact majority and leader election in the graphical setting. We provide simple token-shuffling protocols for these problems, as well as backup protocols to ensure their correctness in all executions.

Our results imply new and improved upper bounds on the time and state complexity of majority and leader election for a wide range of graph families. In some cases, they improve upon the best known upper bounds for these problems. Please see Table 1 for a systematic comparison. While our protocols guarantee *fast* stabilisation in regular graphs with high expansion, they will stabilise in polynomial expected time in *any connected graph.*

Our results suggest the existence of a similar complexity gap in the graphical setting. Specifically, on d-regular graphs with good expansion, such that $d/\beta \in \text{polylog}\, n$, we provide polylogarithmic-time protocols for both leader election and exact majority. This opens a significant complexity gap relative to known constant-state protocols on graphs. For instance, the 4-state exact majority protocol for general graphs [15] requires $\Omega(n)$ parallel time even in regular graphs with high expansion, if node degrees are $\Theta(n)$. Yet, our protocols guarantee stabilisation in only $\text{polylog}\, n$ parallel time in both low and high degree graphs, as long as d/β is at most $\text{polylog}\, n$.

References

1 Dan Alistarh, James Aspnes, and Rati Gelashvili. Space-optimal majority in population protocols. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*. SIAM, 2018. doi:10.1137/1.9781611975031.144.

2 Dan Alistarh and Rati Gelashvili. Recent algorithmic advances in population protocols. *SIGACT News*, 49(3):63–73, 2018. doi:10.1145/3289137.3289150.

3 Dan Alistarh, Rati Gelashvili, and Milan Vojnović. Fast and exact majority in population protocols. In *Proc. 34th ACM Symposium on Principles of Distributed Computing (PODC 2015)*, pages 47–56, 2015.

4 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed computing*, 18(4):235–253, 2006.

5 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

6 James Aspnes and Eric Ruppert. An introduction to population protocols. In *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer, 2009.

7 Joffroy Beauquier, Peva Blanchard, and Janna Burman. Self-stabilizing leader election in population protocols over arbitrary communication graphs. In *International Conference on Principles of Distributed Systems*, pages 38–52. Springer, 2013. URL: `https://hal.archives-ouvertes.fr/hal-00867287v2`.

8 Petra Berenbrink, Tom Friedetzky, Peter Kling, Frederik Mallmann-Trenn, and Chris Wastell. Plurality consensus in arbitrary graphs: Lessons learned from load balancing. In *Proc. 24th Annual European Symposium on Algorithms (ESA 2016)*, volume 57, pages 10:1–10:18, 2016. `doi:10.4230/LIPIcs.ESA.2016.10`.

9 Petra Berenbrink, George Giakkoupis, and Peter Kling. Optimal time and space leader election in population protocols. In *Proc. 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*, pages 119–129, 2020.

10 Pietro Caputo, Thomas M. Liggett, and Thomas Richthammer. Proof of Aldous' spectral gap conjecture. *Journal of the American Mathematical Society*, 23(3):831–851, 2010.

11 Colin Cooper, Tomasz Radzik, Nicolás Rivera, and Takeharu Shiraga. Fast plurality consensus in regular expanders. In *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*, pages 13:1–13:16, 2017. `doi:10.4230/LIPIcs.DISC.2017.13`.

12 Persi Diaconis and Laurent Saloff-Coste. Comparison techniques for random walk on finite groups. *The Annals of Probability*, 21(4):2131–2156, 1993.

13 David Doty, Mahsa Eftekhari, and Eric Severson. A stable majority population protocol using logarithmic time and states, 2020. `arXiv:2012.15800`.

14 David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. *Distributed Computing*, 31(4):257–271, 2018.

15 Moez Draief and Milan Vojnović. Convergence speed of binary interval consensus. *SIAM Journal on Control and Optimization*, 50(3):1087–1109, 2012.

16 Robert Elsässer and Tomasz Radzik. Recent results in population protocols for exact majority and leader election. *Bulletin of the EATCS*, 126, 2018. URL: `http://bulletin.eatcs.org/index.php/beatcs/article/view/549/546`.

17 Leszek Gąsiniec and Grzegorz Stachowiak. Fast space optimal leader election in population protocols. In *Proc. 29th ACM-SIAM Symposium on Discrete Algorithms (SODA 2018)*, 2018. `doi:10.1137/1.9781611975031.169`.

18 Leszek Gąsiniec, Grzegorz Stachowiak, and Przemyslaw Uznański. Time and space optimal exact majority population protocols, 2020. `arXiv:2011.07392`.

19 Johan Jonasson. Mixing times for the interchange process. *Latin American Journal of Probability and Mathematical Statistics*, 9(2):667–683, 2012.

Brief Announcement: How to Trust Strangers – Composition of Byzantine Quorum Systems

Orestis Alpos ✉
University of Bern, Switzerland

Christian Cachin ✉
University of Bern, Switzerland

Luca Zanolini ✉
University of Bern, Switzerland

Abstract

Trust is the basis of any distributed, fault-tolerant, or secure system. A *trust assumption* specifies the failures that a system, such as a blockchain network, can tolerate and determines the conditions under which it operates correctly. In systems subject to Byzantine faults, the trust assumption is usually specified through sets of processes that may fail together. Trust has traditionally been *symmetric*, such that all processes in the system adhere to the same, global assumption about potential faults. Recently, *asymmetric* trust models have also been considered, especially in the context of blockchains, where every participant is free to choose who to trust.

In both cases, it is an open question how to compose trust assumptions. Consider two or more systems, run by different and possibly disjoint sets of participants, with different assumptions about faults: how can they work together? This work answers this question for the first time and offers composition rules for symmetric and for asymmetric quorum systems. These rules are static and do not require interaction or agreement on the new trust assumption among the participants. Moreover, they ensure that if the original systems allow for running a particular protocol (guaranteeing consistency and availability), then so will the joint system. At the same time, the composed system tolerates as many faults as possible, subject to the underlying consistency and availability properties.

Reaching consensus with asymmetric trust in the model of personal Byzantine quorum systems (Losa et al., DISC 2019) was shown to be impossible, if the trust assumptions of the processes diverge from each other. With asymmetric quorum systems, and by applying our composition rule, we show how consensus is actually possible, even with the combination of disjoint sets of processes.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Byzantine quorum systems, composition of quorum systems, trust models, asymmetric trust

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.44

Related Version *Full Version*: https://arxiv.org/abs/2107.11331

Funding This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

1 Extended Abstract

Secure distributed systems rely on *trust*. A security assumption defines the failures and attacks that can be tolerated and names conditions under which the system may operate. Implicitly, this determines the trust in certain components to be correct. In fault-tolerant replicated systems, trust has traditionally been expressed globally, through a *symmetric* assumption on the number or kind of faulty processes, which is shared by all processes. An example of this is the well-known threshold fault assumption: the system tolerates up to a finite and limited number of faulty processes in the system; no guarantees can be given

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 44; pp. 44:1–44:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

beyond this about the correct execution of protocols. More generally, a symmetric trust assumption is defined through a *fail-prone system*, which is a collection of subsets of processes, such that each of them contains all the processes that may at most fail together during a protocol execution.

Quorum systems [14] complement the notion of fail-prone systems and are used within distributed fault-tolerant protocols to express trust assumptions operationally.

In the classical interpretation, a quorum system is a collection of subsets of processes, called *quorums*, with two properties, formally known as *consistency* and *availability*, respectively, that any two quorums have a non-empty intersection and that in every execution, there exists a quorum made of correct processes. *Byzantine quorum systems* (BQS) have been formalized by Malkhi and Reiter [11] and generalize classical quorum systems by tolerating Byzantine failures, i.e., where faulty processes may behave arbitrarily. They are the focus of this work and allow for building secure, trustworthy systems. A BQS assumes one global shared Byzantine fail-prone system and, because of that, use the model of symmetric trust. Consistency for a BQS demands that any two quorums intersect in a set that contains at least one correct process in every execution.

Motivated by the requirements of more flexible trust models, particularly in the context of blockchain networks, new approaches to trust have been explored. It is evident that a common trust model cannot be imposed in an open and decentralized or permissionless environment. Instead, every participant in the system should be free to choose who to trust and who not to trust. Damgård *et al.* [4], and Cachin and Tackmann [2] extend Byzantine quorum systems to permit subjective trust by introducing *asymmetric* Byzantine quorum systems. They let every process specify their own fail-prone system and quorum system. Global system guarantees can be derived from these personal assumptions. Extending traditional Byzantine quorum systems that use threshold assumptions, several recent recent suggestions [7, 6, 10] have also introduced more flexible notions of trust.

In this work, we study the problem of composing trust assumptions, as expressed by symmetric and by asymmetric Byzantine quorum systems. Starting from two or more running distributed systems, each one with its own assumption, how can they be combined, so that their participant groups are joined and operate together? A simple, but not so intriguing solution could be to stop all running protocols and to redefine the trust structure from scratch, with full knowledge of all assumptions across the participants. With symmetric trust, a new global assumption that includes all participants would be defined. In the asymmetric-trust model, every process would specify new personal assumptions on all other participants. Subsequently, the composite system would have to be restarted. Although this solution can be effective, it requires that all members of each initial group express assumptions about the trustworthiness of the processes in the other groups. In realistic scenarios, this might not be possible, since the participants of one system lack knowledge about the members of other systems, and can therefore not express their trust about them. Moreover, one needs to ensure that the combined system satisfies the liveness and safety conditions, as expressed by the B^3-condition for quorum intersection. Since the assumptions are personal, it is not guaranteed, and in practice quite challenging, that the composite system will indeed satisfy the B^3-condition.

Our work, whose details appear in the full paper [1], formulates the problem of composing quorum systems and gives methods for assembling trust assumptions from different, possibly disjoint, systems to a common model. We do so by introducing composition rules for trust assumptions, in both the symmetric-trust and asymmetric-trust model. Our methods describe the resulting fail-prone systems and the corresponding quorum systems.

In a different line of work, subjective trust assumptions have also been introduced with the Stellar blockchain (`https://www.stellar.org`) [13, 8, 9], a cryptocurrency ranked in the top-20 by market capitalization today. In contrast to the original, well-understood notion of quorum systems, these works depart from the classical intersection requirement among quorums. Such systems may fork into separate *consensus clusters*, each one satisfying agreement and liveness on its own. This implies that consensus may hold only "locally", and a unique consensus across disjoint clusters is not possible. More specifically, Losa *et al.* prove [9, Lemma 4] that no quorum-based algorithm can guarantee agreement between two processes whose quorums do not intersect in their model. Our work overcomes this impossibility and shows that consensus can be reached even with disjoint sets of participants, whose trust assumptions do not intersect. Moreover, we use the established notion of quorums, which enables to run many well-understood protocols, such as consensus, reliable broadcast, emulations of shared memory, and more [2, 3].

A related form of recursive composition of (Byzantine) quorum systems has been explored and utilized in the literature. The idea is that, given two systems, each occurrence of a process in the first is *replaced* by a copy of the second system. Malkhi *et al.* [12] construct and study composite BQS, such as *recursive threshold* BQS, using this idea. Hirt and Maurer [5] use this technique to reason about multiparty computation over access structures. Our approach is orthogonal to these works, in the sense that it places the two original systems on the same level. In other words, we explore the failures that two systems can tolerate when they are joined together, as opposed when one is inserted into the other.

In summary, the contributions are as follows [1]:

1. We show how to join together two or more systems in a way where processes in one system do not need a complete knowledge of the trust assumptions of those in the other.
2. We allow processes in each system to maintain their trust assumptions within their original system.
3. We define a deterministic rule to extend the trust assumptions of each system by including the new participants.
4. Our composition rules guarantee that *consistency* and *availability* will be satisfied in the composite quorum system.

References

1 Orestis Alpos, Christian Cachin, and Luca Zanolini. How to trust strangers: Composition of byzantine quorum systems. *CoRR*, abs/2107.11331, 2021. `arXiv:2107.11331`.

2 Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *OPODIS*, volume 153 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

3 Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *CoRR*, abs/2005.08795v3, 2021. `arXiv:2005.08795v3`.

4 Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2007.

5 Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptol.*, 13(1):31–60, 2000.

6 Heidi Howard, Aleksey Charapko, and Richard Mortier. Fast flexible paxos: Relaxing quorum intersection for fast paxos. In *ICDCN*, pages 186–190. ACM, 2021.

7 Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. In *OSDI*, pages 485–500. USENIX Association, 2016.

8 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *SOSP*, pages 80–96. ACM, 2019.

9 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *DISC*, volume 146 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

10 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *CCS*, pages 1041–1053. ACM, 2019.

11 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998.

12 Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. *SIAM J. Comput.*, 29(6):1889–1906, 2000.

13 David Mazières. The Stellar consensus protocol: A federated model for Internet-level consensus. Stellar, available online, `https://www.stellar.org/papers/stellar-consensus-protocol.pdf`, 2016.

14 Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, 1998.

Brief Announcement: Using Nesting to Push the Limits of Transactional Data Structure Libraries

Gal Assa ✉
Technion – Israel Institute of Technology, Haifa, Israel

Hagar Meir ✉
IBM Research, Haifa, Israel

Guy Golan-Gueta ✉
Independent researcher, Israel

Idit Keidar ✉
Technion – Israel Institute of Technology, Haifa, Israel

Alexander Spiegelman ✉
Independent researcher, CA, USA

Abstract

Transactional data structure libraries (TDSL) combine the ease-of-programming of transactions with the high performance and scalability of custom-tailored concurrent data structures. They can be very efficient thanks to their ability to exploit data structure semantics in order to reduce overhead, aborts, and wasted work compared to general-purpose software transactional memory. However, TDSLs were not previously used for complex use-cases involving long transactions and a variety of data structures.

In this paper, we boost the performance and usability of a TDSL, towards allowing it to support complex applications. A key idea is *nesting*. Nested transactions create checkpoints within a longer transaction, so as to limit the scope of abort, without changing the semantics of the original transaction. We build a Java TDSL with built-in support for nested transactions over a number of data structures. We conduct a case study of a complex network intrusion detection system that invests a significant amount of work to process each packet. Our study shows that our library outperforms publicly available STMs twofold without nesting, and by up to 16x when nesting is used.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases Transactional Libraries

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.45

Related Version *Full Version*: https://arxiv.org/abs/2001.00363 [1]

1 Transactional Libraries

The concept of memory transactions [9] is broadly considered to be a programmer-friendly paradigm for writing concurrent code [6, 19]. A transaction spans multiple operations, which appear to execute atomically and in isolation, meaning that either all operations commit and affect the shared state or the transaction aborts. Either way, no partial effects of on-going transactions are observed.

Despite their appealing ease-of-programming, software transactional memory (STM) toolkits [3, 8] are seldom deployed in real systems due to their huge performance overhead. The source of this overhead is twofold. First, an STM needs to monitor all random memory accesses made in the course of a transaction (e.g., via instrumentation in VM-based languages [12]), and second, STMs abort transactions due to conflicts.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 45; pp. 45:1–45:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Instead, programmers widely use concurrent data structure libraries [20, 14, 5, 2], which are much faster but guarantee atomicity only at the level of a single operation on a single data structure.

To mitigate this tradeoff, Spiegelman et al. [21] have proposed *transactional data structure libraries (TDSL)*. In a nutshell, the idea is to trade generality for performance. A TDSL restricts transactional access to a pre-defined set of data structures rather than arbitrary memory locations, which eliminates the need for instrumentation. Thus, a TDSL can exploit the data structures' semantics and structure to get efficient transactions bundling a sequence of data structure operations. It may further manage aborts on a semantic level, e.g., two concurrent transactions can simultaneously change two different locations in the same list without aborting. While the original TDSL library [21] was written in C++, we implement our version in Java.

Quite a few works [13, 23, 15] have used and extended TDSL and similar approaches like STO [10] and transactional boosting [7]. These efforts have shown good performance for fairly short transactions on a small number of data structures. Yet, despite their improved scalability compared to general purpose STMs, TDSLs have also not been applied to long transactions or complex use-cases.

A key challenge arising in long transactions is the high potential for aborts and the large penalty that such aborts induce as much work is wasted.

2 Our Contribution

Transactional nesting. In this paper we push the limits of the TDSL concept in an attempt to make it more broadly applicable. Our main contribution, is facilitating long transactions via *nesting* [17]. Nesting allows the programmer to define nested *child* transactions as self-contained parts of larger *parent* transactions. This controls the program flow by creating *checkpoints*; upon abort of a nested child transaction, the checkpoint enables retrying only the child's part and not the preceding code of the parent. This reduces wasted work, which, in turn, improves performance. At the same time, nesting does not relax consistency or isolation, and continues to ensure that the entire parent transaction is executed atomically. We focus on *closed nesting* [22], which, in contrast to so-called flat nesting, limits the scope of aborts, and unlike open nesting [18], is generic and does not require semantic constructs.

The flow of nesting is shown in Algorithm 1. When a child commits, its local state is migrated to the parent but is not yet reflected in shared memory. If the child aborts, then the parent transaction is checked for conflicts. And if the parent incurs no conflicts in its part of the code, then only the child transaction retries. Otherwise, the entire transaction does. It is important to note that the semantics provided by the parent transaction are not altered by nesting. Rather, nesting allows programmers to identify parts of the code that are more likely to cause aborts and encapsulate them in child transactions in order to reduce the abort rate of the parent.

Yet nesting induces an overhead which is not always offset by its benefits. We investigate this tradeoff using microbenchmarks. We find that nesting is helpful for highly contended operations that are likely to succeed if retried. We also find that nested variants of TDSL improve performance of state-of-the-art STMs with transaction friendly data structures.

NIDS benchmark. We introduce a new benchmark of a *network intrusion detection system (NIDS)* [4], which invests a fair amount of work to process each packet. This benchmark features a pipelined architecture with long transactions, a variety of data structures, and

Algorithm 1 Transaction flow with nesting.

```
TXbegin()
    [Parent code]                         ▷ On abort – retry parent
    nTXbegin()                            ▷ Begin child transaction
        [Child code]                      ▷ On abort – retry child or parent
    nTXend()                              ▷ On commit – migrate changes to parent
    [Parent code]                         ▷ On abort – retry parent
TXend()                                   ▷ On commit – apply changes to thread state
```

multiple points of contention. It follows one of the designs suggested in [4] and executes significant computational operations within transactions, making it more realistic than existing intrusion-detection benchmarks (e.g., [11, 16]).

Enriching the library. In order to support complex applications like NIDS, and more generally, to increase the usability of TDSLs, we enrich our transactional library in with additional data structures – producer-consumer pool, log, and stack – all of which support nesting. The TDSL framework allows us to custom-tailor to each data structure its own concurrency control mechanism. We mix optimism and pessimism (e.g., stack operations are optimistic as long as a child has popped no more than it pushed, and then they become pessimistic), and also fine tune the granularity of locks (e.g., one lock for the whole stack versus one per slot in the producer-consumer pool).

Evaluation. We evaluate our library using our NIDS application, and compare it against existing general purpose STMs. We find that nesting can improve performance by up to 8x. Moreover, nesting improves scalability, reaching peak performance with as many as 40 threads as opposed to 28 without nesting.

References

1 Gal Assa, Hagar Meir, Guy Golan-Gueta, Idit Keidar, and Alexander Spiegelman. Using nesting to push the limits of transactional data structure libraries. *arXiv preprint*, 2021. `arXiv:2001.00363`.

2 Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45. ACM, 2010.

3 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*. Springer, 2006.

4 Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA*. Springer, 2004.

5 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*. Springer, 2005.

6 Maurice Herlihy. The transactional manifesto: software engineering and non-blocking synchronization. In *PLDI 2005*. ACM, 2005.

7 Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

8 Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*. ACM, 2003.

9 Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

10 Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware transactions for faster concurrent code. In *Eurosys*, 2016.

11 Guy Korland. Jstamp, 2014. URL: `https://github.com/DeuceSTM/DeuceSTM/tree/master/src/test/jstamp`.

12 Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with java STM. In *MULTIPROG*, 2010.

13 Pierre LaBorde, Lance Lebanoff, Christina Peterson, Deli Zhang, and Damian Dechev. Wait-free dynamic transactions for linked data structures. In *PMAM*, 2019.

14 Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

15 Lance Lebanoff, Christina Peterson, and Damian Dechev. Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2019.

16 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, 2008.

17 John Eliot Blakeslee Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MIT Cambridge lab, 1981.

18 Yang Ni, Vijay S Menon, Ali-Reza Adl-Tabatabai, Antony L Hosking, Richard L Hudson, J Eliot B Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

19 Michael Scott. Transactional memory today. *SIGACT News*, 46(2), 2015.

20 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS 2000*. IEEE, 2000.

21 Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. Transactional data structure libraries. In *PLDI 2016*. ACM, 2016.

22 Alexandru Turcu, Binoy Ravindran, and Mohamed M Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

23 Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. Lock-free transactional transformation for linked data structures. *TOPC*, 5(1), 2018.

Brief Announcement: Twins – BFT Systems Made Robust

Shehar Bano ✉
Facebook Novi, London, UK

Alberto Sonnino ✉
Facebook Novi, London, UK

Andrey Chursin ✉
Facebook Novi, Menlo Park, CA, USA

Dmitri Perelman ✉
Facebook Novi, Menlo Park, CA USA

Zekun Li ✉
Facebook Novi, Menlo Park, CA, USA

Avery Ching ✉
Facebook Novi, Menlo Park, CA, USA

Dahlia Malkhi ✉
Diem Association, Wilmington, DE, USA
Facebook Novi, Menlo Park, CA, USA

Abstract

Twins is an effective strategy for generating test scenarios with *Byzantine* [10] nodes in order to find flaws in Byzantine Fault Tolerant (BFT) systems. Twins finds flaws in the design or implementation of BFT protocols that may cause correctness issues. The main idea of Twins is the following: running *twin* instances of a node that use correct, unmodified code and share the same network identity and credentials allows to emulate most interesting Byzantine behaviors. Because a twin executes normal, unmodified node code, building Twins only requires a thin wrapper over an existing distributed system designed for Byzantine tolerance. To emulate material, interesting scenarios with Byzantine nodes, it instantiates one or more twin copies of the node, giving the twins the same identities and network credentials as the original node. To the rest of the system, the node and all its twins appear indistinguishable from a single node behaving in a "questionable" manner. This approach generates many interesting Byzantine behaviors, including equivocation, double voting, and losing internal state, while forgoing uninteresting behavior scenarios that can be filtered at the transport layer, such as producing semantically invalid messages.

Building on configurations with twin nodes, Twins systematically generates scenarios with Byzantine nodes via enumeration over protocol rounds and communication patterns among nodes. Despite this being inherently exponential, one new flaw and several known flaws were materialized by Twins in the arena of BFT consensus protocols. In all cases, protocols break within fewer than a dozen protocol rounds, hence it is realistic for the Twins approach to expose the problems. In two of these cases, it took the community more than a decade to discover protocol flaws that Twins would have surfaced within minutes. Additionally, Twins has been incorporated into the continuous release testing process of a production setting (DiemBFT [7]) in which it can execute 44M Twins-generated scenarios daily.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Distributed Systems, Byzantine Fault Tolerance, Real-World Deployment

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.46

Related Version *Full Version*: `https://arxiv.org/abs/2004.10617`

Funding This work is funded by Novi, a subsidiary of Facebook.

Acknowledgements The authors would like to thank Ben Maurer, David Dill, Daniel Xiang, Kartik Nayak, and Ling Ren for feedback on late manuscript, and George Danezis for comments on early manuscript. We also thank the Novi Research and Engineering teams for valuable feedback.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 46; pp. 46:1–46:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 The Twins Approach

Twins systematically constructs test scenarios in which some nodes have one or more twins, and the adversary can delay and drop messages between nodes, i.e., the communication is asynchronous. Twins scenarios are constructed with logical protocol rounds. For each round, the scenario indicates which nodes have twins and which nodes can be reached by other nodes. In addition, each round can designate which nodes are acting as *leaders*, which is a common role in BFT protocols. Executing Twins scenarios requires a thin shim layer that emulates message scheduling and delivery and has a handle to designate a protocol leader.

In notation, nodes are represented by capital alphabets letters (e.g., A) and the twin of a node is represented by the same letter with the prime symbol (e.g., A'). Nodes acting in leader roles are underlined, e.g., $\underline{A}$. We denote partitions of nodes by sets P_*, as in $P_1 = \{A, B, C, D\}$, $P_2 = \{E, F, G\}$. For example, a single-round scenario in which a leader equivocates in the first round and partitions the system into two sets of nodes, each getting a different proposal, can be described as follows:

- Set up a system with nodes $\{D, D', E, F, G\}$.
- Initialize D and D' with different inputs v_1 and v_2.
- Execute round 1 with partitions $P_1 = \{\underline{D}, E, G\}$, $P_2 = \{\underline{D'}, F\}$.

Although enumerating round-by-round scenarios is inherently exponential, experience shows that protocols with logical flaws break with a handful of nodes in less than a dozen rounds (see e.g., [1]). Indeed, the full paper shows several succinct Twins scenarios that expose known BFT protocol flaws, as well as a scenario that surfaces a flaw in a recent protocol that hasn't been exposed before. Of these, we chose to present below one Twins scenario. It demonstrates that in Tendermint [4] and Casper [5], a leader must delay the maximal transmission bound; removing this delay would break liveness.

2 Preliminaries: PBFT, Tendermint and Casper

The goal of BFT replication is for a group of nodes to provide a fault-tolerant service through redundancy. Clients submit requests to the service. These requests are collectively sequenced by the nodes; this enables all nodes to execute the same chain of requests and hence agree on their (deterministic) output. Practical Byzantine Fault Tolerance (PBFT) [6] is a hallmark work that was designed to work efficiently in the asynchronous setting. Carrying the classical PBFT solution to the blockchain world, Tendermint [4] and Capser [5] introduced a much simplified *linear* strategy for leader-replacement. However, it has been observed [3, 12] that this strategy forgoes an important property of asynchronous protocols – *Responsiveness* – the ability of a leader to advance as soon as it receives messages from $2f + 1$ nodes.[1] We demonstrate that this delay in fact mandatory: if the leader's delay was removed from Tendermint (equiv Casper), the protocol would lose liveness. .

3 Example: A Flawed Tendermint Variant

In a nutshell, the flawed variant works as follows. A quorum certificate *(QC)* is formed on a leader proposal if it gathers $2f + 1$ votes from nodes. A leader proposes to extend the highest QC it knows. Nodes vote on the leader proposal if it extends the highest QC they

[1] Tendermint is a precursor to HotStuff [13] and DiemBFT [7] which operates in two-phase views, but has no Responsiveness. HotStuff/DiemBFT solve this by adding a third phase.

know. A commit decision on the leader proposal forms if it gathers $2f+1$ votes forming a QC, and then $2f+1$ nodes vote for that QC. Progress is hinged on leaders obtaining the highest QC in the system, otherwise liveness is broken.

We demonstrate through a Twins scenario that liveness is broken. Lack of progress is detected by observing that two consecutive views with honest leaders whose communication with a quorum is timely do not produce a decision.

The liveness-attack scenario uses 4 replicas (D, E, F, G), where D has a twin D'. In the first view, D and D' generate equivocating proposals. Only D, E receive a QC for D's proposal. The next leader is F who re-proposes the proposal by D', which E and D do not vote for because they already have a QC for that height. Only F and D' receive a QC for F's proposal. This scenario repeats itself indefinitely, resulting in loss of liveness. More specifically, this scenario works as follows:

View 1: Initialize D and D' with different inputs v_1 and v_2.

- Create the partitions $P_1 = \{\underline{D}, E, G\}$, $P_2 = \{\underline{D'}, F\}$.
- Let D and D' run as leaders for one round. D proposes v_1 to P_1 and gathers votes from P_1 creating QC(v_1). D' proposes v_2 to P_2 and gathers votes but not a QC.
- Create the following partitions: $P_1 = \{D, E\}$, $P_2 = \{\underline{D'}, F\}$, $P_3 = \{\underline{G}\}$. D broadcasts QC(v_1), which only reaches P_1 i.e., (D, E).

View 2: Drop all proposals from D and D' until View 2 starts.

- Remove all partitions, i.e., $P = \{D, D', E, \underline{F}, G\}$.
- Let F run as leader for one round. F re-proposes v_2 (i.e., D''s proposal in the previous round) to P. (D, E) do not vote as they already have QC(v_1) for that height. F gathers votes from the other nodes and forms QC(v_2).
- Create partitions $P_1 = \{D, E\}$, $P_2 = \{\underline{D'}, F\}$, $P_3 = \{\underline{G}\}$.
- F broadcasts QC(v_2), which only reaches P_2.

View 3: Drop all proposals from F until View 3 starts.

- Create the partitions $P_1 = \{D, \underline{E}, G\}$, $P_2 = \{\underline{D'}, F\}$.
- Let E run as leader for one round. E proposes v_3 which extends the highest QC it knows, QC(v_1). As before, E manages to form Q(v_3), but as a result of a partition, the QC will only reach (D, E). Next, there is a view-change, F is the new leader, and there are no partitions. F proposes v_4 which extends QC(v_2), the highest QC it knows. However, (D, E) do not vote because v_4 does not extend their highest QC i.e., QC(v_3). This scenario can repeat itself indefinitely, resulting in the loss of liveness.

4 What Else?

The full version of the paper presents a new flaw exposed by Twins in Fast HotStuff [8] and known flaws re-materialized as Twins scenarios in several BFT protocols (Zyzzyva [9], FaB [11], Sync HotStuff [2]). In all cases, exposing vulnerabilities requires only a small number of nodes, partitions, rounds and leader rotations. We implemented an automated scenario generator for Twins and show that our implementation covers the described scenarios within minutes.

References

1 Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance: Thelma, Velma, and Zelma. arXiv preprint, 2018. arXiv:1801.10022.

2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. In *IEEE Symposium on Security and Privacy*, 2020.

3 Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. `https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf`, 2016.

4 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The Latest Gossip on BFT Consensus. arXiv preprint, 2018. `arXiv:1807.04938`.

5 Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. arXiv preprint, 2017. `arXiv:1710.09437`.

6 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation*, 1999.

7 Diem. DiemBFT. `https://github.com/diem/diem`.

8 Mohammad M Jalalzai, Jianyu Niu, and Chen Feng. Fast-hotstuff: A fast and resilient hotstuff protocol. arXiv preprint, 2020. `arXiv:2010.11454`.

9 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

10 Leslie Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

11 J-P Martin and Lorenzo Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

12 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus in the Lens of Blockchain. arXiv preprint, 2018. `arXiv:1803.05069`.

13 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT Consensus with Linearity and Responsiveness. In *ACM Symposium on Principles of Distributed Computing*, 2019.

Brief Annoucement: On Extending Brandt's Speedup Theorem from LOCAL to Round-Based Full-Information Models

Paul Bastide
Ecole Normale Supérieure de Rennes, France

Pierre Fraigniaud
Université de Paris and CNRS, France

Abstract

Given any task Π, Brandt's speedup theorem (PODC 2019) provides a mechanical way to design another task Π' on the same input-set as Π such that, for any $t \geq 1$, Π is solvable in t rounds in the LOCAL model if and only if Π' is solvable in $t-1$ rounds in the LOCAL model. We dissect the construction in Brandt's speedup theorem for expressing it in the broader framework of all round-based models supporting full information protocols, which includes models as different as asynchronous wait-free shared-memory computing with iterated immediate snapshots, and synchronous failure-free network computing.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Local Checkability, Distributed Complexity and Computability

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.47

Related Version *Full Version*: https://arxiv.org/abs/2108.01989

Acknowledgements Additional supports from ANR Project DUCAT.

1 Introduction

Given a complexity or computability result established for a distributed computing model $\mathcal{M}_1$, several questions can be raised. Does this result hold for another model $\mathcal{M}_2$? What makes this result true for $\mathcal{M}_1$ but not for $\mathcal{M}_2$, or what are the features common to $\mathcal{M}_1$ and $\mathcal{M}_2$ that make the result true for both models? For instance, if a result holds in the LOCAL model [4,5], is it because the model is synchronous? Is it because processes and communication links are failure-free? Is it because the network satisfies some property (e.g., large girth)? Is it because the problem satisfies some property (e.g., local checkability)? A typical example is Brandt's speedup theorem [3]. This theorem essentially provides a mechanical way to construct a task Π' from any task Π, on the same input set as Π, such that, for every $t \geq 1$, Π is solvable in t rounds in LOCAL if and only if Π' is solvable in $t-1$ rounds in LOCAL. This theorem is an efficient tool for designing lower bounds. Indeed, starting from a task Π, iterating the construction results in a series of tasks $\Pi^{(r)}, r \geq 1$, such that, for every $t \geq 1$, Π is solvable in t rounds if and only if $\Pi^{(r)}$ is solvable in $t-r$ rounds. In particular, $\Pi^{(t)}$ is solvable in zero rounds, and demonstrating that $\Pi^{(t)}$ is actually not solvable in zero rounds establishes the lower bound $t+1$ for the round-complexity of Π.

Brandt's speedup theorem does not directly applies to LOCAL, but to an anonymous variant of LOCAL on graphs with sufficiently large *girth*. This is because the presence of identifiers assigned to the nodes prevents *local-independence* to be satisfied, where the latter is a property that is essential for establishing the theorem. It is not trivial to formally express this property, but, roughly speaking, given the radius-$(t-1)$ views of two adjacent nodes v and v' in some network G, the presence of identifiers results in the fact that one cannot guarantee that two independent extensions of these two views into radius-t views are compatible. Local independence also imposes to consider graphs G with girth $g > 2t-1$. Indeed, in graphs with girth $g \leq 2t-1$, two independent radius-t extensions of the radius-$(t-1)$ views of v and v' may include a same node w provided with different identifiers, or with different inputs. This would result into two non-compatible radius-t extensions in the sense that there are no instances yielding the simultaneous presence of these two radius-t views at two adjacent nodes. Also, Brandt's speedup theorem requires the tasks at hand to be *locally checkable*. This property essentially says that, given an assignment of input-output values to the nodes, the correctness of the collection of output values with respect to the collection of input values can be established by merely inspecting the values of each node and of its neighbors in the network. In other words, a task is locally checkable if the correctness of an assignment of values to the nodes is defined as the conjunction of the local correctness of this assignment, where "local" refers to the closed neighborhood of each node. Proper coloring and maximal independent set (MIS) are typical examples of locally checkable tasks in LOCAL.

We can now rephrase our original questioning in the specific case of Brandt's speedup theorem: does this theorem holds in other models? For such a question to make sense, we restrict attention to models in which the notion of *rounds* is defined, which naturally include synchronous models in networks with multiparty interactions, namely *hypergraphs*, and synchronous models in networks that evolve with time, namely, *dynamic networks*. Round-based models however include far more than just synchronous models in networks. For instance, asynchronous shared-memory computing with *iterated immediate snapshots*, referred to as WAIT-FREE in the following, which is computationally equivalent to asynchronous read/write shared-memory computing with crash-prone processes, is round-based. The same holds for t-resilient computing, $0 \leq t \leq n-1$, which is essentially the same as WAIT-FREE, but where at most t processes can crash [1]. The LOCAL model has another feature. It supports *full information* communication protocols. That is, whenever a process receives information from another process, one can assume that the latter has sent *all* the data it acquired before the communication took place. This assumption enables the design of strong lower bounds, which hold even if the processes are not restricted in term of volume of communication. Also, the LOCAL model does not restrict the individual computational power of the processes. This assumption enables the design of unconditional lower bounds, which hold independently from complexity or computability assumptions regarding the computing power of each individual process. All the models mentioned above support full-information protocols, and have unlimited individual computational power.

So, making our questioning even more specific: Is there an analog of Brandt's speedup theorem for all round-based models supporting full-information protocols with unlimited individual computational power? If not, what make the LOCAL model so special? If yes, for which models? Under which conditions?

2 Our Results

We refer to [2] for a complete description of our results. Using the framework provided by combinatorial topology applied to distributed computing, we give a general definition of speedup tasks for round-based models supporting full-information protocols (see Fig. 1). Given a task Π in the LOCAL model, Brandt's speedup theorem constructs such a speedup task $\Pi' = \Phi(\Pi)$. We then revisit Brandt's construction, that is, we dissect the nature of the operator Φ transforming any task Π into a task $\Pi' = \Phi(\Pi)$, for identifying the central assumptions allowing this construction to work in LOCAL.

$$\begin{tikzcd}
\mathcal{P}^{(t-1)} \arrow[d, "\alpha"'] & \mathcal{I} \arrow[l, "\Xi^{t-1}"'] \arrow[r, "\Xi^{t}"] \arrow[dl, "\Delta'"'] \arrow[dr, "\Delta"] & \mathcal{P}^{(t)} \arrow[d, "\delta"] \\
\mathcal{O}' \arrow[r, "\Xi"] & \Xi(\mathcal{O}') \arrow[r, "\beta"] & \mathcal{O}
\end{tikzcd}$$

Figure 1 The task $\Pi' = (\mathcal{I}, \mathcal{O}', \Delta')$ is a speedup task for $\Pi = (\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ and $\mathcal{O}$ denote the input and output complexes, respectively, and Δ denotes the input-output specification. Ξ denotes the map corresponding to the communication model $\mathcal{M}$ at hand, and $\mathcal{P}^{(t)}$ denotes the protocol complex at round t. All maps α, β, and δ are simplicial.

They are two central assumptions in Brandt's construction: *local checkability* and *local independence*. We extend these two notions from the LOCAL model to round-based models supporting full-information protocols. We also extend Brandt's operator Φ to all such models. We denote by $\Phi^\star$ this extension. As a result, we are able to express a general speedup theorem, which roughly reads as follows.

▶ **Theorem 1.** *Let $\mathcal{M}$ be a round-based model supporting full-information protocols, let Π be a task, and let $t \geq 1$. The task $\Phi^\star(\Pi)$ satisfies the following:*

1. *Assume that Π satisfies $(t-1)$-independence with respect to $\mathcal{M}$. If Π is solvable in at most t rounds, then $\Phi^\star(\Pi)$ is solvable in at most $t-1$ rounds.*
2. *Assume that Π is locally checkable in $\mathcal{M}$. If $\Phi^\star(\Pi)$ is solvable in at most $t-1$ rounds, then Π is solvable in at most t rounds.*

Statement 1 guarantees that the task $\Phi^\star(\Pi)$ is at least "1-round faster" than the original task Π. Note that that local independence is actually sufficient for deriving lower bounds. Statement 2 guarantees that $\Phi^\star(\Pi)$ is no more than "1-round faster", and, in particular, that $\Phi^\star(\Pi)$ is not a "trivial" task. Observe that the sets of hypotheses required for each of the two statements are different. Concretely, our general construction $\Phi^\star$ allows us to directly extend Brandt's speedup theorem to various kinds of synchronous models in networks, including directed graphs, hypergraphs, dynamic networks, and even to graphs including short cyclic dependencies between processes (i.e., small girth). Interestingly, our general construction also enables to extend Brandt's speedup theorem to asynchronous failure-prone computing models such as WAIT-FREE. In particular, we provide a new impossibility proof for consensus and for perfect renaming in 2-process systems. The case of consensus is an example of a non locally checkable task for which our approach still provide a non-trivial lower bound.

3 Conclusion

Our construction $\Phi^\star$ is based on identifying specific subcomplexes of the output complex, for generalizing Brandt's construction. This approach is well suited to LOCAL, and to its extensions to hypergraphs and dynamic networks. However, it does not satisfactorily match the characteristics of WAIT-FREE for large systems, essentially because WAIT-FREE does not satisfy the local independence property whenever $n > 2$. Nevertheless, we conjecture that there might be another way to decompose the output complex into subcomplexes that would provide a speedup theorem for models not satisfying local independence (e.g., WAIT-FREE), but this decomposition still remains to be found.

References

1 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Series on Parallel and Distributed Computing. Wiley, 2004.

2 Paul Bastide and Pierre Fraigniaud. On extending Brandt's speedup theorem from LOCAL to round-based full-information models, 2021. arXiv:2108.01989.

3 Sebastian Brandt. An automatic speedup theorem for distributed problems. In *38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 379–388, 2019. doi:10.1145/3293611.3331611.

4 Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992. doi:10.1137/0221015.

5 David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Discrete Mathematics and Applications. SIAM, 2000.

Brief Announcement: Automating and Mechanising Cutoff Proofs for Parameterized Verification of Distributed Protocols

Shreesha G. Bhat ✉
Indian Institute of Technology Madras, India

Kartik Nagar ✉
Indian Institute of Technology Madras, India

Abstract

We propose a framework to automate and mechanize simulation-based proofs of cutoffs for parameterized verification of distributed protocols. We propose a strategy to derive the simulation relation given the cutoff instance and encode the correctness of the simulation relation as a formula in first-order logic. We have successfully applied our approach on a number of distributed protocols.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Formal Methods, Automated Verification, Distributed Protocols

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.48

Related Version *Full Version*: `https://github.com/shreesha00/DISC2021-Brief-Announcement-Automating-and-Mechanising-Cutoff-Proofs-for-Parameterized-Verification`

Funding This work was supported by the Young Research Fellow Program at IITM.

1 Introduction

The problem of parameterized verification [1] of distributed protocols asks whether a protocol satisfies its specification for all values of the parameter. Here, the parameter is typically the number of nodes involved in the protocol. Cutoff based approaches for parameterized verification rely on the following observation: If the protocol can break its specification for some value of the parameter, it is guaranteed to break the specification for a value $\leq k$, where k is also called the cutoff. Small cutoffs can then enable fully automated verification, for example by exhaustively model checking all instances of the protocol of size $\leq k$.

In the recent past, there has been a lot of interest in automated and mechanised verification of distributed protocols [11, 4, 12, 7, 9, 10]. Most of these approaches rely on constructing and proving some form of inductive invariant. While previous works have also attempted to use cut-off based approaches for verification [3, 6, 8, 1], they have mostly been limited to either a restricted class of protocols [6] or a restricted class of specifications [8]. In this work, we develop a methodology for mechanising simulation-based proofs of cutoffs by observing that the simulation relation can be generated using a direct correspondence between the nodes of an arbitrarily large system and the cutoff instance. We have successfully applied the proposed approach on a variety of distributed protocols.

2 Proposed Technique with Example

Model. We consider distributed protocols modelled in RML [11] where the system state is represented by a set of relations. The communication model between nodes is assumed to be asynchronous message passing. A set of actions are defined with guards and each step of the protocol involves non-deterministically firing one of these actions in an atomic fashion. The

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 48; pp. 48:1–48:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

specification for the protocol is given as a safety property. The parameterized verification problem then asks whether for all instances of the protocol, does the specification hold at every step. We are interested in a cutoff on the number of nodes. As an example protocol, we consider Leader Election in a Ring as given in [11]. The system consists of a finite number of nodes in a ring setting. Each node has a unique ID, and there is a total order on the IDs. There are two actions, (1) $\texttt{generate}(N, ID(N), NG(N))$: Node N sends a message $ID(N)$ to its neighbour $NG(N)$ and (2) $\texttt{handle\_message}(N, m, NG(N))$: Node N takes a pending message m in its pending queue (denoted by `pnd`) and forwards it to its neighbour only if $m > ID(N)$, else if $m = ID(N)$, N is elected as a leader (denoted by `leader`). The specification for the protocol is that there is at most one leader.

Inputs. We assume that the protocol designer provides the proposed framework with, (1) The protocol description, (2) A cutoff instance, (3) A mapping from nodes of any arbitrary system to nodes of the cutoff instance given by sim, (4) Two functions Ω and τ as described below. Let C be the cutoff instance and $\mathbb{M}_C$ the set of nodes in C. Consider an arbitrary instance L of the protocol where $\mathbb{M}_L$ is the set of nodes in L such that $|\mathbb{M}_L| > |\mathbb{M}_C|$. The mapping function sim has the following meaning, for each node $N_L \in \mathbb{M}_L$, N_L is simulated by $sim(N_L) \in \mathbb{M}_C$. The key intuition here is that a node $N_C \in \mathbb{M}_C$ effectively maintains the state components relevant to the violation of the safety property for all nodes $N_L \in \mathbb{M}_L$ such that $sim(N_L) = N_C$. To show that $|\mathbb{M}_C|$ is the cutoff, we will show that for some sequence of actions which leads to the first violation of the specification in any instance L of size $|\mathbb{M}_L| > |\mathbb{M}_C|$, there also exists a sequence of actions in the cutoff system of size $|\mathbb{M}_C|$ which leads to a violation. Specifically, for the example of Leader Election, for any arbitrary size system L with at least two nodes where nodes L_A and L_B (IDs A and B) are elected as leaders, we consider a cutoff system of size 2 with nodes C_A and C_B with the same IDs. The sim function is such that nodes in the portion of the ring in-between L_A and L_B (in the direction of communication) including L_B are simulated by C_B and the rest of the nodes in L are simulated by C_A.

Simulation Relation. Our observation is that a generic form of the simulation relation can be given in-terms of the sim function. Let σ_L and σ_C denote the states of two instances L and C respectively. We can in general view σ_X as a function from nodes of the instance to state components. The simulation R between states maintains the property that all effects of actions that can contribute to a violation and are present in the state of a node in L must be present in the state of its simulating node in C. This can be mathematically stated as follows:

$$(\sigma_L, \sigma_C) \in R \Leftrightarrow \forall N \in \mathbb{M}_L.\, \Omega(\sigma_L(N)) \subseteq \sigma_C(sim(N))$$

The relation uses the function Ω, which filters out those state components (i.e. effects of actions) which do not contribute in any way to the violation of the specification. With respect to our example of Leader Election, the above general strategy translates to the following,

$$\begin{aligned}(\sigma_L, \sigma_C) \in R \iff \forall N \in \mathbb{M}_L.\, & (\texttt{leader}_L(L_A) \rightarrow \texttt{leader}_C(C_A)) \wedge (\texttt{leader}_L(L_B) \rightarrow \texttt{leader}_C(C_B)) \\ & \wedge (\neg\texttt{leader}_L(L_A) \wedge \texttt{pnd}_L(A, N) \rightarrow \texttt{pnd}_C(A, sim(N))) \\ & \wedge (\neg\texttt{leader}_L(L_B) \wedge \texttt{pnd}_L(B, N) \rightarrow \texttt{pnd}_C(B, sim(N)))\end{aligned}$$

Lock Step. The lock-step describes the action(s) taken in C for every action taken in L. The generic strategy behind the lock-step is that actions involving any two nodes L_1 and L_2 in L are translated to actions involving $sim(L_1)$ and $sim(L_2)$ in C. Note that this might result in some steps where $sim(L_1) = sim(L_2)$, which represents a *stuttering step* where L transitions according to the action but C stays in the same state. Stuttering steps can also occur when L performs some action that C cannot perform. Similarly, a single action in L might need to be simulated by more than one action in C. This behaviour can be encapsulated in a function τ. In general, action a in L is translated to $\tau(a)$ in C, where $\tau(a)$ can be a sequence of zero or more actions where zero actions represents a stuttering step. In our example of leader election, the lockstep relation τ is defined as follows:

$$\begin{aligned}\tau(\texttt{generate}_L(L_A, A, NG(L_A))) &= \texttt{generate}_C(C_A, A, C_B)\\ \tau(\texttt{handle\_message}_L(L_A, A, NG(L_A))) &= \texttt{handle\_message}_C(C_A, A, C_B)\\ \tau(\texttt{handle\_message}_L(L_A, B, NG(L_A))) &= \texttt{handle\_message}_C(C_A, B, C_B)\texttt{generate}_C(C_B, B, C_A).\end{aligned}$$

Note that the second action is required to maintain the simulation relation. Apart from these, the symmetric versions with A and B interchanged are also included in the lockstep.

FOL Encoding & Theorem. To prove that the simulation relation holds at each step, we show that it is an inductive invariant of the combined instances L and C. Given FOL encoding of the states and actions of the protocol, we construct the following FOL formula to check the correctness of the simulation relation:

$$(\sigma_L, \sigma_C) \in R \wedge a(\sigma_L, \sigma'_L) \wedge \tau(a)(\sigma_C, \sigma'_C) \wedge (\sigma'_L, \sigma'_C) \notin R \tag{1}$$

Here, a can be any of the actions possible in L according to the protocol description, and we use the notation $a(\sigma_X, \sigma'_X)$ to denote the change in state after the action.

We construct a FOL encoding consisting of the protocol states, actions and the simulation relation along with the formula 1. If the resulting formula is UNSAT, then the simulation relation holds at every step. Note that we are only interested in the first violation of the specification in any arbitrary instance, because once a single violation occurs, the specification is broken and the protocol is incorrect, therefore, if Φ denotes the specification, we also conjunct $\Phi(\sigma_L)$ and $\Phi(\sigma_C)$ to the above formula. We also need to show that violations would be preserved by the simulation relation:

$$\neg\Phi(\sigma_L) \wedge R(\sigma_L, \sigma_C) \wedge \Phi(\sigma_C) \tag{2}$$

▶ **Theorem 1.** *If the formulae 1 and 2 are unsatisfiable, and if the cutoff instance C does not violate the specification Φ, then no instance of the protocol violates Φ.*

3 Experiments and Future Work

The tool implementing the ideas described in the paper is still a work-in-progress. However, we have some positive preliminary results: we have been able to verify the leader election protocol and the significantly more complicated sharded key-value store protocol [4]. We use Z3 [2] as a back-end SMT solver, and in both cases, the verification time is in the order of a few seconds. In addition, we have manually checked the correctness of our approach on a variety of other protocols: Lock Service [13], Learning Switch [11], Distributed Lock Service [5]. As part of future work, we plan to finish the tool and apply our method on more complex protocols. In addition, we also want to leverage our observation regarding the relation between the cutoff instance and violations of the specification to automatically synthesize the cutoff instance.

To conclude, in this work, we have proposed an approach to significantly simplify and automate cut-off based proofs for verification of distributed protocols. Our experience is that cutoff-based proofs can be applied to a large number of distributed protocols, and we hope that this work would pave the way for more widespread application of this proof technique.

References

1 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

2 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

3 E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94. ACM Press, 1995.

4 Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 405–425. Springer, 2019.

5 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17. ACM, 2015.

6 Nouraldin Jaber, Swen Jacobs, Christopher Wagner, Milind Kulkarni, and Roopsha Samanta. Parameterized verification of systems with global synchronization and guards. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 299–323. Springer, 2020.

7 Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: incremental inference of inductive invariants for verification of distributed protocols. In *SOSP*, pages 370–384. ACM, 2019.

8 Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 217–237. Springer, 2017.

9 Kenneth L. McMillan and Oded Padon. Ivy: A multi-modal verification tool for distributed algorithms. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 190–202. Springer, 2020.

10 Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.*, 1(OOPSLA):108:1–108:31, 2017.

11 Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630. ACM, 2016.

12 Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. In *PLDI*, pages 662–677. ACM, 2018.

13 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368. ACM, 2015.

Brief Announcement: Local Certification of Graph Decompositions and Applications to Minor-Free Classes

Nicolas Bousquet
Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Laurent Feuilloley
Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Théo Pierron
Univ. Lyon, Université Lyon 1, LIRIS UMR CNRS 5205, F-69621, Lyon, France

Abstract

Local certification consists in assigning labels to the nodes of a network to certify that some given property is satisfied, in such a way that the labels can be checked locally. In the last few years, certification of graph classes received a considerable attention. The goal is to certify that a graph G belongs to a given graph class $\mathcal{G}$. Such certifications with labels of size $O(\log n)$ (where n is the size of the network) exist for trees, planar graphs and graphs embedded on surfaces. Feuilloley et al. ask if this can be extended to any class of graphs defined by a finite set of forbidden minors.

In this paper, we develop new decomposition tools for graph certification, and apply them to show that for every small enough minor H, H-minor-free graphs can indeed be certified with labels of size $O(\log n)$. We also show matching lower bounds with a new simple proof technique.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Local certification, proof-labeling schemes, locally checkable proofs, graph decompositions, minor-free graphs

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.49

Related Version *Full Version*: `https://arxiv.org/abs/2108.00059` [2]

Funding This work was supported by ANR project GrR (ANR-18-CE40-0032).

1 Introduction

Local certification is an active field of research in the theory of distributed computing. On a high level it consists in certifying global properties in such a way that the verification can be done locally. More precisely, for a given property, a local certification consists of a labeling (called a *certificate assignment*), and of a local verification algorithm. If the configuration of the network is correct, then there should exist a labeling of the nodes that is accepted by the verification algorithm, whereas if the configuration is incorrect no labeling should make the verification algorithm accept.

Local certification originates from self-stabilization, and was first concerned with certifying that a solution to an algorithmic problem is correct. However, it is also important to understand how to certify properties of the network itself, that is, to find locally checkable proofs that the network belongs to some graph class. There are several reasons for that. First, because certifying some solutions can be hard in general graphs, while they become simpler on more restricted classes. To make use of this fact, it is important to be able to certify that the network does belong to the restricted class. Second, because some distributed algorithms work only on some specific graph classes, and we need a way to ensure that the network does

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 49; pp. 49:1–49:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

belong to the class, before running the algorithm. Third, the distinction between certifying solutions and network properties is rather weak, in the sense that the techniques are basically the same. So we should take advantage of the fact that a lot is known about graph classes to learn more about certification.

In the domain of graph classes certification, there have been several results on various classes such as trees [10], bipartite graphs [9] or graphs of bounded diameter [3], but until two years ago little was known about essential classes, such as planar graphs, H-free or H-minor-free graphs. Recently, it has been shown that planar graphs and graphs of bounded genus can be certified with $O(\log n)$-bit labels [7, 8, 5]. This size, $O(\log n)$, is the gold standard of certification, in the sense that little can be achieved with $o(\log n)$ bits, thus $O(\log n)$ is often the best we can hope for. It happens that planar and bounded-genus graphs are classic examples of graphs classes defined by forbidden minors, which naturally raises the following question.

▶ **Question 1** ([8, 6]). *Can any graph class defined by a finite set of forbidden minors be certified with $O(\log n)$-bit certificates?*

This open question is quite challenging: there are as many good reasons to believe that the answer is positive as negative.

First, the literature provides some reasons to believe that the conjecture is true. Properties that are known to be hard to certify, that is, that are known to require large certificates, are very different from minor-freeness. Specifically, all these properties (*e.g.* small diameter [3], non-3-colorability [9], having a non-trivial automorphism [9]) are non-hereditary. That is, removing a node or an edge may yield a graph that is not in the class. Intuitively, hereditary properties might be easier to certify in the sense that one does not need to encode information about every single edge or node, as the class is stable by removal of edges and nodes. Minor-freeness is a typical example of hereditary property. Moreover, this property, that has been intensively studied in the last decades, is known to carry a lot of structure, which is an argument in favor of the existence of a compact certification (that is a certification with $O(\log n)$-bit labels).

On the other hand, from a graph theory perspective, it might be surprising that a general compact certification existed for minor-free graphs. Indeed, for the known results, obtaining a compact certification is tightly linked to the existence of a precise constructive characterization of the class (*e.g.* a planar embedding for planar graphs [7, 5], or a canonical path to the root for trees [10]). While such a characterization is known for some restricted minor-closed classes, we are far from having such a characterization for every minor-closed class. Note that there are a lot of combinatorial and algorithmic results on H-minor free graphs, but they actually follow from properties satisfied by H-minor free graphs, not from exact characterizations of such graphs. For certification, we need to rule out the graphs that do not belong to the class, hence a characterization is somehow necessary.

It is important to note that forbidden minor characterizations are about structures that are absent from the graphs, and local certification is often about certifying the existence of some structures, which explains why it is a challenge to certify all minor-free classes with small certificates. On the other hand, as we will see later, certifying that a minor does appear in the graph is easy.

1.1 Our results

An extensive line of work in structural graph theory aims to provide characterizations of classes using so-called decomposition theorems. Amongst the most famous examples of these theorems is the proof of the 4-Color Theorem [1] or the Strong Perfect Graph Theorem [4] which consists in decomposing graphs until we reach some elementary graphs.

Our goal in this paper is to prove that many of these decomposition tools can actually be used in order to certify the fact that the graph belongs to the class. We first prove that several classic decomposition techniques existing in the literature are suitable for certification. We then apply these general tools on Question 1 to prove that several H-minor free graph classes can be certified with $O(\log n)$ bits. In particular, our results provide evidence that, if the answer to Question 1 is negative then it is certainly for large non-planar graphs H.

The decomposition tools we are able to certify are at the core of many decomposition theorems: 2-(edge-)connectivity, 3-connectivity, block-cut trees, forbidden subgraphs, and expansions of nodes or edges by new graphs. Note that these tools are also interesting by themselves, in particular, connectivity is an important measure of robustness in networks. One common challenge in the design of certification for decomposition is what we call *certificate congestion*. Consider for example a situation in which we have a certification for k graphs, and we want to merge these graphs by identifying one vertex in each of them. Then, the straightforward technique to certify the merged graph is to give to the merged node its certificate for every of these k graphs. But since k might be large, this implies a large certificate size. Since we aim for small certificates, we want to avoid such congestion. We use several solutions to cope with this problem.

Using these tools, we show that the answer to Question 1 is positive for many small graphs. These results permit to illustrate our methods with simple and compact applications of our tools. More generally, we aim at providing some evidence that graph decomposition and these tools can be successfully used in the certification setting, and it is very likely that many other minor-closed classes can be certified using our techniques.

Our main results are summarized in Figure 1, and illustrations of the corresponding minors can be found in Figure 2. These are actually the hard cases of the following theorem.

▶ **Theorem 2.** *H-minor-free classes can be certified in $O(\log n)$ bits when H has at most 4 vertices.*

We also prove a general $\Omega(\log n)$ lower bounds for H-minor-freeness for all 2-connected graphs H. This generalizes and simplifies the lower bounds of [7] which apply only to K_k and $K_{p,q}$-minor-free graphs, and use ad-hoc and more complicated techniques.

Class	Optimal size	Result
K_3-minor free	$\Theta(\log n)$	Equivalent to acyclicity [10, 9].
Diamond-minor-free	$\Theta(\log n)$	New.
K_4-minor-free	$\Theta(\log n)$	New.
$K_{2,3}$-minor-free	$\Theta(\log n)$	New.
$(K_{2,3}, K_4)$-minor-free (*i.e.* outerplanar)	$\Theta(\log n)$	New.
$K_{2,4}$-minor-free	$\Theta(\log n)$	New.

Figure 1 Our main results for the certification of minor-closed classes.

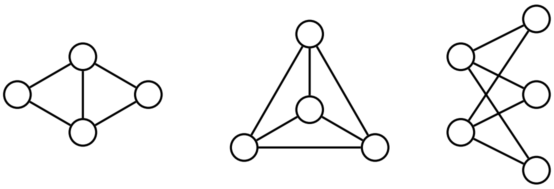


Figure 2 From left to right: the diamond, the clique on 4 vertices K_4, and the complete bipartite graph $K_{2,3}$.

References

1 Kenneth Appel, Wolfgang Haken, et al. Every planar map is four colorable. *Bulletin of the American mathematical Society*, 82(5):711–712, 1976.

2 Nicolas Bousquet, Laurent Feuilloley, and Théo Pierron. Local certification of graph decompositions and applications to minor-free classes. *CoRR*, abs/2108.00059, 2021. arXiv:2108.00059.

3 Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate proof-labeling schemes. *Theor. Comput. Sci.*, 811:112–124, 2020. doi:10.1016/j.tcs.2018.08.020.

4 Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas. The strong perfect graph theorem. *Annals of mathematics*, pages 51–229, 2006.

5 Louis Esperet and Benjamin Lévêque. Local certification of graphs on surfaces. *CoRR*, abs/2102.04133, 2021. arXiv:2102.04133.

6 Laurent Feuilloley. Introduction to local certification. *CoRR*, abs/1910.12747, 2019.

7 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Éric Rémila, and Ioan Todinca. Compact distributed certification of planar graphs. In *PODC '20: ACM Symposium on Principles of Distributed Computing*, pages 319–328. ACM, 2020. doi:10.1145/3382734.3404505.

8 Laurent Feuilloley, Pierre Fraigniaud, Pedro Montealegre, Ivan Rapaport, Eric Rémila, and Ioan Todinca. Local certification of graphs with bounded genus. *CoRR*, abs/2007.08084, 2020.

9 Mika Göös and Jukka Suomela. Locally checkable proofs in distributed computing. *Theory of Computing*, 12(19):1–33, 2016. doi:10.4086/toc.2016.v012a019.

10 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.

Brief Announcement: Memory Efficient Massively Parallel Algorithms for LCL Problems on Trees

Sebastian Brandt
ETH Zürich, Switzerland

Rustam Latypov
Aalto University, Finland

Jara Uitto
Aalto University, Finland

Abstract

We establish scalable Massively Parallel Computation (MPC) algorithms for a family of fundamental graph problems on trees. We give a general method that, for a wide range of LCL problems, turns their message passing counterparts into exponentially faster algorithms in the sublinear MPC model. In particular, we show that any LCL on trees that has a deterministic complexity of $O(n)$ in the LOCAL model can be sped up to $O(\log n)$ (high-complexity regime) in the sublinear MPC model and similarly $n^{o(1)}$ to $O(\log \log n)$ (intermediate-complexity regime). We emphasize, that we work on bounded degree trees and all of our algorithms work in the sublinear MPC model, where local memory is $O(n^{\delta})$ for $\delta < 1$ and global memory is $O(m)$.

For the high-complexity regime, one key ingredient is a novel *pointer-chain* technique and analysis that allows us to solve any solvable LCL on trees with a sublinear MPC algorithm with complexity $O(\log n)$. For the intermediate-complexity regime, we adapt the approach by Chang and Pettie [FOCS'17], who gave a canonical algorithm for solving LCL problems on trees in the LOCAL model. For the special case of 3-coloring trees, which is a natural LCL problem, we provide a conditional $\Omega(\log \log n)$ lower bound, implying that solving LCL problems on trees with deterministic LOCAL complexity $n^{o(1)}$ requires $\Theta(\log \log n)$ deterministic time in the sublinear MPC model when using a natural family of component-stable algorithms.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed computing, Locally checkable labeling problems, Trees, Massively Parallel Computation, Sublinear memory, 3-coloring

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.50

Funding *Rustam Latypov*: Supported in part by the Academy of Finland, Grant 334238.

1 Introduction

Many fundamental graph problems in the study of distributed and parallel graph algorithms fall under the umbrella of locally checkable labeling (LCL) problems, where each node can locally check whether the output is locally correct. Classic examples include graph colorings, MIS, maximal matching, distributed Lovász Local Lemma (LLL), and many more. These problems serve as abstractions to fundamental primitives in large-scale graph processing and have recently gained a lot of attention [6, 7, 8, 9].

In this work, we study LCL problems in the sublinear Massively Parallel Computation (MPC) model introduced by Karloff et al. [11]. The MPC model is a mathematical abstraction of modern data processing platforms such as MapReduce, Hadoop, Spark, and Dryad. In this model, we have M machines who communicate in an all-to-all fashion. We focus on problems where the input is modeled as a graph of n vertices and m edges. Initially, the graph is divided among the M machines. Each machine has n^{δ} *local memory*, where the units of memory are *words* of $O(\log n)$ bits. When $\delta < 1$, the memory regime is often referred

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 50; pp. 50:1–50:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to as sublinear (or strongly sublinear [3, 4]). The number of machines is chosen such that $n^\delta \cdot M = \Omega(m)$. Ideally, the total memory $M \cdot n^\delta$ is linear in or only slightly higher than the input size m. In this work, we restrict ourselves to linear global memory, and note that, for the $O(\log n)$ regime, speeding up everything exponentially in bounded degree trees with superlinear $O(m^{1+\delta})$ global memory would be straightforward using the well-known graph exponentiation technique [12]. For simplicity, we assume that each vertex is hosted on an independent virtual machine and the memory restriction is that no virtual machine should use more than $O(n^\delta)$ memory. A crucial challenge that comes with the linear global memory restriction is that only a small fraction of $n^{1-\delta}$ of these virtual machines can simultaneously use up their maximum local memory.

1.1 Related Work

In the last decade, there has been tremendous progress in understanding the complexities of locally checkable problems in various models of distributed and parallel computing.

As the most relevant examples to our work, we want to highlight two recent papers. In the randomized/deterministic LOCAL and CONGEST model, Balliu et al. [1] showed that the possible complexity classes of locally checkable problems in rooted regular trees are fully understood. In the CONGEST model, Balliu et al. [2] showed that on trees, the complexity of an LCL problem is asymptotically equal to its complexity in the LOCAL model. They also showed that the same does not hold in general graphs. It is worth noting that in order to prove the asymptotic equality between LOCAL and CONGEST for LCL problems on trees, Balliu et al. [2] use very similar, but independently developed, techniques to what we use in our intermediate-complexity regime.

Regarding our coloring result, it is worth mentioning that in the low-memory setting an $O(\log \log n)$-round *randomized* MPC algorithm for 4-coloring trees was given by Ghaffari, Grunau, and Jin [9]. For the classical $(\Delta + 1)$-coloring problem the deterministic state of the art is given by a very recent result by Czumaj, Davies, and Parter providing an $O(\log \log \log n)$-round algorithm [7].

2 Results

Our main results are twofold: we show that any LCL on trees that has a complexity of $O(n)$ in the LOCAL model can be sped up to $O(\log n)$ (high-complexity regime) in the sublinear MPC model and similarly $n^{o(1)}$ to $O(\log \log n)$ (intermediate-complexity regime). For the LCL problem of 3-coloring trees, we provide a conditional $\Omega(\log \log n)$ lower bound for component-stable algorithms. This implies that solving LCL problems on trees with LOCAL complexity $n^{o(1)}$ requires $\Theta(\log \log n)$ time in the sublinear MPC model by component-stable algorithms. The lower bound is conditioned on a widely believed conjecture and currently, the family of component-stable algorithms rules out the known techniques for coloring trees with few colors. Next, we introduce the formal statements and a high level idea of the techniques that we used to obtain our results; the proof details and a more thorough discussion of related work will appear in the full version of the paper.

▶ **Theorem 1** (High-complexity regime). *Every LCL on constant degree trees that admits a correct solution can be solved deterministically in $O(\log n)$ rounds with $O(n^\delta)$ words of local memory for any constant $\delta > 0$ and $O(m)$ words of global memory in the MPC model.*

The proof of Theorem 1 is constructive: we explicitly provide, for any solvable LCL, an algorithm $\mathcal{A}$ that has a runtime of $O(\log n)$. On a high level, algorithm $\mathcal{A}$ proceeds in 3 phases. The first phase consists in rooting the input tree by using a Rake & Compress style process that also orients the removed paths. In the second phase, roughly speaking, the goal is to compute, for a substantial number of nodes v, the set of output labels that can be output at v such that the label choice can be extended to a (locally) correct solution in the subtree hanging from v. This is done in an iterative manner, proceeding from the leaves towards the root. The last phase consists in using the computed information to solve the given LCL from the root downwards.

While this outline sounds simple, there are a number of intricate challenges that require the development of novel techniques, both in the design of the algorithm and its analysis: for instance, the depth of the input tree can be much larger than $\Theta(\log n)$ (which prevents us from performing the above ideas in a sequential manner even when using a Rake & Compress process, and introduces a new challenge in the form of interleaving Rake & Compress steps), and the storage of the required completability information in a standard implementation exceeds the available memory even when using graph exponentiation. One of our key technical contributions is the design of a fine-tuned potential function for the analysis of the complex algorithm resulting from addressing these issues.

▶ **Theorem 2** (Intermediate-complexity regime). *Consider an LCL problem Π with a deterministic LOCAL complexity $n^{o(1)}$. Then, there is a deterministic MPC algorithm that solves Π in time $O(\log\log n)$ with $O(n^{\delta})$ words of local memory for any constant $\delta > 0$ and $O(m)$ words of global memory.*

The proof is similar to the proof of Lemma 14 by Chang and Pettie [5]. Essentially, we simulate their LOCAL algorithm but pay special attention to the memory usage. Let us outline the modifications required to reach our goal.

First, we adopt their Rake & Compress decomposition for trees by synthesizing an exponentially faster algorithm that computes the decomposition in the MPC model. Using this decomposition we divide the nodes into *batches* according to which layer (or partition) they belong to such that removing one batch from the decomposition reduces the number of nodes by a factor of Δ (maximum degree of the graph). We process the graph one batch at a time for $O(\log\log n)$ phases until we have enough global memory to directly simulate the LOCAL algorithm in a constant number of rounds. During each phase we simulate the LOCAL algorithm for the lowest batch, after which we perform graph exponentiation.

▶ **Theorem 3** (Conditional hardness). *The problem of 3-coloring constant degree trees in the sublinear MPC model has deterministic complexity $O(\log\log n)$. Under the connectivity conjecture, there is no $o(\log\log n)$ round component-stable MPC algorithm for 3-coloring of constant degree trees.*

Our conditional lower bound is based on the existence of high-girth graphs that are not 3-colorable by Marshal [13], and on the work of Ghaffari et al. [10], which assumes component-stability and conditions on the widely believed connectivity conjecture in MPC. The family of component-stability algorithms captures all the currently known methods to color trees with 3 colors. In a very recent work, Czumaj et al. [6] show that some problems can be solved much faster with component-unstable algorithms than with stable ones. While their results cast uncertainty on the strength of our lower bound, we note that it is not clear at all how to extend the separation result to problems such as the 3-coloring that relies on "global" graph properties such as the arboricity. Furthermore, their algorithms require much more global memory than what we allow in our paper.

References

1 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studeny, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *PODC*, 2021. arXiv:2102.09277.

2 Alkida Balliu, Keren Censor-Hillel, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Locally checkable labelings with small messages. In *DISC*, 2021. arXiv:2105.05574.

3 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Matching and MIS for uniformly sparse graphs in the low-memory MPC model. *Theorerical Computer Science*, 2018. arXiv:1807.05374.

4 Sebastian Brandt, Manuela Fischer, and Jara Uitto. Breaking the linear-memory barrier in MPC: Fast MIS on trees with strongly sublinear memory. In *SIROCCO*, 2019. doi:10.1007/978-3-030-24922-9_9.

5 Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the LOCAL model. In *FOCS*, 2017. doi:10.1109/FOCS.2017.23.

6 Artur Czumaj, Peter Davies, and Merav Parter. Component stability in low-space massively parallel computation. In *PODC*, 2021. arXiv:2106.01880.

7 Artur Czumaj, Peter Davies, and Merav Parter. Improved deterministic $(\Delta + 1)$ coloring in low-space MPC. In *PODC*, 2021. doi:10.1145/3465084.3467937.

8 Michal Dory, Orr Fischer, Seri Khoury, and Dean Leitersdorf. Constant-round spanners and shortest paths in Congested Clique and MPC. In *PODC*, 2021. doi:10.1145/3465084.3467928.

9 Mohsen Ghaffari, Christoph Grunau, and Ce Jin. Improved MPC algorithms for MIS, matching, and coloring on trees and beyond. In *DISC*, 2020. arXiv:2002.09610.

10 Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. In *FOCS*, 2019. doi:10.1109/FOCS.2019.00097.

11 Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *SODA*, 2010. doi:10.1137/1.9781611973075.76.

12 Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *PODC*, 2010. doi:10.1145/1835698.1835772.

13 Simon Marshall. Another simple proof of the high girth, high chromatic number theorem. *The American Mathematical Monthly*, 2008. doi:10.1080/00029890.2008.11920498.

Brief Announcement: Revisiting Signature-Free Asynchronous Byzantine Consensus

Christian Cachin ✉
University of Bern, Switzerland

Luca Zanolini ✉
University of Bern, Switzerland

Abstract

Among asynchronous, randomized, and signature-free implementations of consensus, the protocols of Mostéfaoui et al. (PODC 2014 and JACM 2015) represent a landmark result, which has been extended later and taken up in practical systems. The protocols achieve optimal resilience and take, in expectation, only a constant expected number of rounds and have quadratic message complexity. Randomization is provided through a common-coin primitive. However, the first version of this simple and appealing protocol suffers from a little-known liveness issue due to asynchrony. The JACM 2015 version avoids the problem, but is considerably more complex.

This work revisits the original protocol of PODC 2014 and points out in detail why it may not progress. A fix for the protocol is presented, which does not affect any of its properties, but lets it regain the original simplicity in asynchronous networks enhanced with a common-coin protocol.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Randomized consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.51

Related Version *Full Version*: https://arxiv.org/pdf/2005.08795.pdf

Funding This work has been funded by the Swiss National Science Foundation (SNSF) under grant agreement Nr. 200021_188443 (Advanced Consensus Protocols).

1 Introduction

Consensus is a fundamental abstraction in distributed systems. It captures the problem of reaching agreement among multiple processes on a common value, despite unreliable communication and the presence of faulty processes. Consensus in asynchronous networks requires randomization.

Mostéfaoui et al. [4] presented a signature-free round-based asynchronous consensus algorithm for binary values at PODC 2014. It had received considerable attention because it was the first asynchronous consensus protocol with optimal resilience, tolerating up to $f < \frac{n}{3}$ Byzantine processes, that did not use digital signatures. Hence, it needs only authenticated channels and remains secure against a computationally unbounded adversary. Moreover, it takes $O(n^2)$ constant-sized messages in expectation. (This description excludes the necessary cost for implementing randomization, for which the protocol relies on an abstract common-coin primitive, as defined by Rabin [6].) The algorithm represents a landmark result, and practical systems, such as "Honey Badger BFT" [3], have later taken it up, and many others have extended it.

However, this protocol, which we call the *PODC-14* version [4] in the following, suffers from a subtle and little-known problem. It may violate liveness, i.e., an adversary can prevent progress among the correct processes by controlling the messages between them and by sending them values in a specific order. This has explicitly pointed out by Tholoniat and Gramoli [7] in 2019. Indeed, the corresponding journal publication by Mostéfaoui et al. [5], to which we refer as the *JACM-15* version, only touches on the issue and goes on to present a modified, extended protocol. This fixes the problem, but requires also many more communication steps and adds considerable complexity.

In this work, we introduce an improved protocol that changes the PODC-14 version in a simple, but crucial way and thereby regains the simplicity of the original result. The reader may find detailed descriptions and arguments in the full version [2].

2 The Byzantine consensus algorithm of PODC-14

Let us briefly recall the consensus algorithm from the PODC-14 version (Alg. 1). A correct process may propose a binary value b by invoking *rbc-propose*(b); the consensus abstraction decides for b through an *rbc-decide*(b) event. The algorithm proceeds in rounds. In each round, an instance of *bv-broadcast* is invoked, a primitive introduced in the same paper [4, Figure 1]. A correct process p_i executes *bv-broadcast* and waits for a value b to be delivered, identified by a tag characterizing the current round. When such a bit b is received, p_i adds b to *values* and broadcasts b through an AUX message to all processes. Whenever a process receives an AUX message containing b from p_j, it stores b in a local set $aux[j]$. Once p_i has received a set $B \subseteq values$ of values such that every $b \in B$ has been delivered in AUX messages from at least $n - f$ processes, then p_i releases the coin for the round. Subsequently, the process waits for the coin protocol to output a binary value s through *output-coin*(s), tagged with the current round number.

Process p_i then checks if there is a single value b in B. If so, and if $b = s$, then it decides for value b. The process then proceeds to the next round with proposal b. If there is more than one value in B, then p_i changes its proposal to s. In any case, the process starts another round and invokes a new instance of *bv-broadcast* with its proposal.

Algorithm 1 Randomized binary consensus according to Mostéfaoui et al. [4] (code for p_i).

```
State
    $round \leftarrow 0$: current round
    $values \leftarrow \{\}$: set of bv-delivered binary values for the round
    $aux \leftarrow [\{\}]^n$: stores sets of values that have been received in AUX messages in the round
upon event rbc-propose($b$) do
    invoke bv-broadcast($b$) with tag $round$
upon bv-deliver($b$) with tag $r$ such that $r = round$ do
    $values \leftarrow values \cup \{b\}$
    send message [AUX, $round, b$] to all $p_j \in \mathcal{P}$
upon receiving a message [AUX, $r, b$] from $p_j$ such that $r = round$ do
    $aux[j] \leftarrow aux[j] \cup \{b\}$
upon exists $B \subseteq values$ such that $B \neq \{\}$ and $|\{p_j \in \mathcal{P} \mid B = aux[j]\}| \geq n - f$ do
    release-coin with tag $round$
    wait for output-coin($s$) with tag $round$
    $round \leftarrow round + 1$
    if exists $b$ such that $B = \{b\}$ then                         // i.e., $|B| = 1$
        if $b = s$ then
            output rbc-decide($b$)
        invoke bv-broadcast($b$) with tag $round$            // propose $b$ for the next round
    else
        invoke bv-broadcast($s$) with tag $round$ // propose coin value $s$ for the next round
    $values \leftarrow [\bot]^n$; $aux \leftarrow [\{\}]^n$
```

3 The problem and a solution

In the problematic execution [7], the network reorders messages between correct processes and delays them until the coin value becomes known. Our crucial insight concerns the coin: In any full implementation, it is not abstract, but implemented by a protocol that exchanges messages among the processes. Based on this, our solution consists of two parts.

Our *first* change is to assume FIFO ordering on the reliable point-to-point links, including the messages exchanged by the coin implementation. FIFO-ordered links are actually a very common assumption. They are easily implemented by adding sequence numbers to messages [1]. Our *second* change is to allow the set B (Alg. 2, line 25) to dynamically change while the coin protocol executes. Alg. 2 implements these changes. More details and correctness proofs appear in the full version [2].

Algorithm 2 Randomized binary consensus (code for p_i).

```
 1: State
 2:     $round \leftarrow 0$: current round
 3:     $values \leftarrow \{\}$: set of bv-delivered binary values for the round
 4:     $aux \leftarrow [\{\}]^n$: stores sets of values that have been received in AUX messages in the round
 5:     $decided \leftarrow []^n$: stores binary values that have been reported as decided by other processes
 6:     $sentdecide \leftarrow$ FALSE: indicates whether $p_i$ has sent a DECIDE message
 7: upon event $rbc\text{-}propose(b)$ do
 8:     invoke $bv\text{-}broadcast(b)$ with tag $round$
 9: upon $bv\text{-}deliver(b)$ with tag $r$ such that $r = round$ do
10:     $values \leftarrow values \cup \{b\}$
11:     send message $[\text{AUX}, round, b]$ to all $p_j \in \mathcal{P}$
12: upon receiving a message $[\text{AUX}, r, b]$ from $p_j$ such that $r = round$ do
13:     $aux[j] \leftarrow aux[j] \cup \{b\}$
14: upon receiving a message $[\text{DECIDE}, b]$ from $p_j$ such that $decided[j] = \bot$ do
15:     $decided[j] = b$
16: upon exists $b \neq \bot$ such that $|\{p_j \in \mathcal{P} \mid decided[j] = b\}| \geq f + 1$ do
17:     if $\neg sentdecide$ then
18:         send message $[\text{DECIDE}, b]$ to all $p_j \in \mathcal{P}$
19:         $sentdecide \leftarrow$ TRUE
20: upon exists $b \neq \bot$ such that $|\{p_j \in \mathcal{P} \mid decided[j] = b\}| \geq 2f + 1$ do
21:     $rbc\text{-}decide(b)$
22:     halt
23: upon exist $|Q_i = \{p_j \in \mathcal{P} \mid aux[j] \subseteq values\}| \geq 2f + 1$ do
24:     $release\text{-}coin$ with tag $round$
25: upon event $output\text{-}coin(s)$ with tag $round$ and $\exists B \neq \{\}, \forall\, p_j \in Q_i : B = aux[j]$ do
26:     $round \leftarrow round + 1$
27:     if exists $b$ such that $|B| = 1 \wedge B = \{b\}$ then
28:         if $b = s \wedge \neg sentdecide$ then
29:             send message $[\text{DECIDE}, b]$ to all $p_j \in \mathcal{P}$
30:             $sentdecide \leftarrow$ TRUE
31:         invoke $bv\text{-}broadcast(b)$ with tag $round$                // propose $b$ for the next round
32:     else
33:         invoke $bv\text{-}broadcast(s)$ with tag $round$ // propose coin value $s$ for the next round
34:     $values \leftarrow [\bot]^n$; $aux \leftarrow [\{\}]^n$
```

References

1 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

2 Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *CoRR*, abs/2005.08795v3, 2021. `arXiv:2005.08795v3`.

3 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. ACM CCS*, pages 31–42, 2016.

4 Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proc. PODC*, pages 2–9, 2014.

5 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *J. ACM*, 62(4):31:1–31:21, 2015.

6 Michael O. Rabin. Randomized byzantine generals. In *Proc. FOCS*, pages 403–409, 1983.

7 Pierre Tholoniat and Vincent Gramoli. Formal verification of blockchain byzantine fault tolerance. In *6th Workshop on Formal Reasoning in Distributed Algorithms (FRIDA'19)*, 2019.

Brief Announcement: Non-Blocking Dynamic Unbounded Graphs with Worst-Case Amortized Bounds

Bapi Chatterjee
Institute of Science and Technology, Klosterneuburg, Austria

Sathya Peri
Indian Institute of Technology, Hyderabad, India

Muktikanta Sa
Télécom SudParis – Institut Polytechnique de Paris, France

Abstract

This paper reports a new concurrent graph data structure that supports updates of both edges and vertices and queries: Breadth-first search, Single-source shortest-path, and Betweenness centrality. The operations are provably linearizable and non-blocking.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases concurrent data structure, linearizability, non-blocking, directed graph, breadth-first search, single-source shortest-path, betweenness centrality

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.52

Related Version *Full Version*: `https://arxiv.org/abs/2003.01697`

Funding This work was partially funded by National Supercomputing Mission, Govt. of India under the project "Concurrent and Distributed Programming primitives and algorithms for Temporal Graphs"(DST/NSM/R&D_Exascale/2021/16).

1 Introduction

Dynamic graph data structures with concurrent query operations and updates can readily boost important real-world applications such as social networks [6], semantic-web [5], biological networks [10], blockchains [3], and many others. The existing libraries of graph queries, which support dynamic updates, for example, Stinger [12], GraphOne [16], GraphTinker [15], Kineograph [9], GraphTau [14], Kickstarter [18], Aspen [11], etc. face limitations such as blocking concurrency, no native support for vertex updates, and high memory-footprint.

In this paper, we describe the design and implementation of a graph data structure, which provides (a) three useful operations – breadth-first search (BFS), single-source shortest-path (SSSP), and betweenness centrality (BC), (b) dynamic updates of edges and vertices concurrent with the operations, (c) non-blocking progress with linearizability [13], and (d) a light memory footprint. We call it PANIGRAHAM [a]: **Pra**ctical **N**on-blocki**ng** **Gra**ph Algorit**hms**. In a nutshell, we implement a concurrent non-blocking dynamic directed graph data structure as an *adjacency-list* formed by a composition of lock-free sets: a lock-free hash-table and multiple lock-free binary search trees (BSTs). The set of outgoing edges E_v from a vertex $v \in V$ is implemented by a BST, whereas, v itself is a node of the hash-table. Addition/removal of a vertex translates to the same operation in the lock-free hash-table,

[a] Panigraham is the Sanskrit translation of Marriage, which undoubtedly is a prominent event in our lives resulting in networks represented by graphs.

```
Operation Op(v)
  tid ← GetThId();// get thread-id
  if (isMrkd(v)) then
    return NULL; //Vertex is not present
  return Scan(v, tid);//Invoke Scan
Method Scan(v, tid)
  list⟨SNode⟩ ot, nt ; //Trees to hold the nodes
  ot ← TreeCollect (v, tid); //$1^{st}$ Collect
  while (true) do //Repeat the tree collection
    nt ← TreeCollect (v, tid); //$2^{nd}$ Collect
    if (CmpTree (ot, nt)) then
      return nt;//return if two collects are equal
    ot ← nt;
Method CmpTree(ot, nt)
  if ($ot$ = NULL ∨ $nt$ = NULL) then
    return false;
  oit ← ot.head, nit ← nt.head;
  while ($oit \neq ot$.tail ∧ $nit \neq nt$.tail ) do
    if ($oit$.n ≠ $nit$.n ∨ $oit$.ecnt ≠ $nit$.ecnt ∨ $oit$.p ≠ $nit$.p) then
      return false; //Both the trees are not equal
    oit ← oit.nxt; nit ← nit.nxt;
  if ($oit$.n ≠ $nit$.n ∨ $oit$.ecnt ≠ $nit$.ecnt ∨ $oit$.p ≠ $nit$.p) then //Both the trees are not equal
    return false ;
  else return true ; //Both the trees are equal
Method ChkVisit(adjn, tid, count)
  if (adjn.oi.VisA [tid] = count) then
    return true;
  else return false ;
Method TreeCollect(v, tid)
  queue ⟨SNode ⟩ que; //Queue used for traversal
  list⟨SNode ⟩st; cnt ←cnt + 1; //List to keep of the visited nodes
  v.oi.VisA [tid] ← cnt;
  sn←new CTNode(v,NULL,NULL, v.oi.ecnt);//Create a new SNode
  st.Add(sn);que.enque(sn);
  while (¬que.empty()) do //Iterate all vertices
    cvn ← que.deque(); // Get the front node
    if (isMrkd (cvn)) then
      continue;// If marked then continue
    itn ← cvn.n.enxt; //Get the root ENode
    stack ⟨ENode ⟩ S; // stack for inorder traversal
    /*Process all neighbors of cvn in the order of
    inorder traversal, as the edge-list is a BST*/
    while (itn ∨ ¬S.empty()) do
      while (itn ) do
        if (¬isMrkd(itn)) then
          S.push(itn); // push the ENode
        itn ← itn.el;
      itn ← S.pop();
      if (¬isMrkd(itn)) then //Validate it
        adjn ← itn.ptv;
        if (¬isMrkd (adjn)) then //Validate it
          if (¬ChkVisit (adjn, tid, cnt)) then
            adjn.oi.VisA [tid] ← cnt; //Mark it
            //Create a new SNode
            sn ← new CTNode(adjn, cvn,NULL,adjn.oi.ecnt);
            st.Add(sn); //Insert sn to st
            que.enque(sn); //Push sn into the que
      itn ← itn.er;
  return st; //The tree is returned to the Scan
```

Figure 1 Framework interface operation for graph queries.

whereas, addition/removal of an edge translates to the same operation in a lock-free BST. The operations – BFS, SSSP, BC – are implemented by specialized partial snapshots. In a dynamic concurrent setting, we apply multi-scan/validate [1] to ensure the *linearizability* of a partial snapshot. We prove that these operations are *non-blocking*. The empirical results show the effectiveness of our algorithms.

2 PANIGRAHAM

Algorithm Overview. We implement an ADT $\mathscr{A} = \mathscr{S} \cup \mathscr{Q}$, wherein the set operations $\mathscr{S} := \{\text{PutV}, \text{RemV}, \text{GetV}, \text{PutE}, \text{RemE}, \text{GetE}\}$ use lock-free hash-table and BST and the queries $\mathscr{Q} := \{\text{BFS}, \text{SSSP}, \text{BC}\}$ use partial snapshot. To de-clutter the presentation, we encapsulate the three queries in a unified framework with an interface operation Op–presented in pseudo-code in Figure 1. Op is specialized to the requirements of the three queries. We have explained the pseudo-code using line-comments in Figure 1. For detail of the ADT operations please see the full version [8], wherein we also present their proofs of linearizability and non-blocking progress.

Experimental Results and Discussion. We experimentally evaluate our non-blocking graph against two well-known existing batch analytics methods: (a) **Stinger** [12], and (b) **Ligra** [17]. To analyze the trade-off between consistency and performance, in addition to the presented linearizable algorithm PANIGRAHAM (**PG-Cn**), we include its inconsistent variant (**PG-Icn**). The results are based on a standard dataset **R-MAT** graphs [7]. Each micro-benchmark displays the latency of an end-to-end run of 10^4 operations on a loaded graph, assigned in a uniform random order to the threads. We used a range of workload distributions. A sample label, say, 2/49/49 on the top of a column of performance plots refers to a distribution $\{\textsc{Op} : 2\%, \{\textsc{PutV} : 24.5\%, \textsc{RemV} : 24.5\%\}, \{\textsc{PutE} : 24.5\%, \textsc{RemE} : 24.5\%\}\}$. We used a multi-core system with 28 cores (56 logical threads). The results shown in Figure 2 demonstrate the scalability of the proposed methods. We observe that the presented algorithm outperforms **Stinger** and **Ligra** in several cases by orders of magnitude. In the full version [8] we present additional results on real-life graphs as well as experimental comparison of the memory-footprints of the methods that further highlights the efficacy of our method.

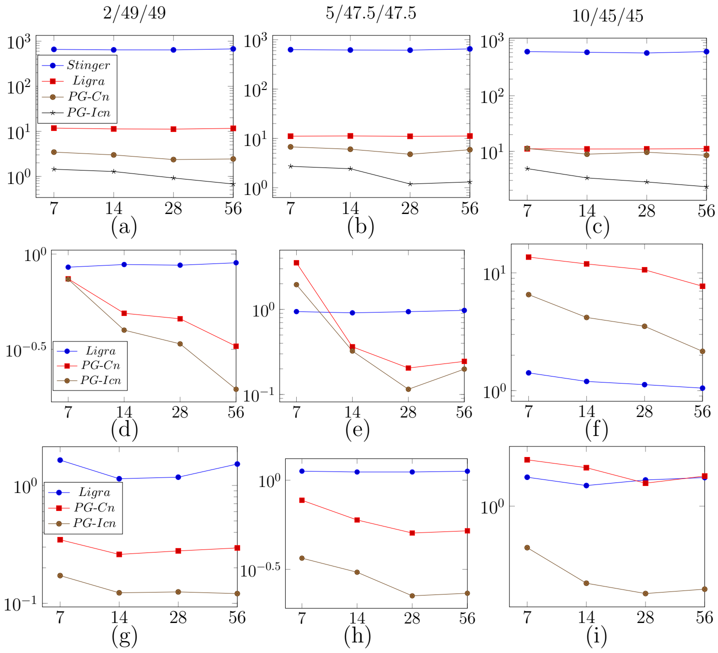


Figure 2 Latency of the executions containing $\textsc{Op}$: (1) BFS ((a), (b), and (c)) on a graph of size $|V|= 131K$ and $|E|= 2.44M$, (2) SSSP ((d), (e), and (f)) on a graph of size $|V|= 8K$ and $|E|= 80K$, and (3) BC ((g), (h), and (i)) on a graph of size $|V|= 16K$ and $|E|= 160K$. X-axis and Y-axis units are the number of threads and time in second, respectively.

Complexity Analysis. Given a graph $G = (V, E)$, denote $|V|= n$, $|E|= m$, $\max_{\mathtt{v}\in V}(\delta_{\mathtt{v}}) = \delta$, where $\delta_{\mathtt{v}}$ is the degree of vertex $\mathtt{v}$. Define the (static) *state* of a graph G as a tuple $S_G = (n, m, \delta)$. Let X be a concurrent execution given as a set of operations. Thus, for an $o \in X$, $type(o) \in \mathscr{A}$, where $type(o)$ denotes the type of o and $\mathscr{A}$ is the ADT. Let I_o and C_o be the *interval contention* [2] and *point contention* [4], respectively, for an $o \in X$. Denote $\widetilde{I_o} = (I_o - 1)$, the total number of concurrent operation calls *other than o itself* (those responsible for a possible cost escalation) that were invoked between the invocation and response of o. Denote the worst-case cost of o, given o is invoked at an atomic time point when state of G was S_G by W_{o,S_G}. W_{o,S_G} for each operation type is given in Table 1 of [8]. The states of G, being tuples, are ordered by dictionary order. In a dynamic setting, W_{o,S_G} is upper-bounded by the worst-case cost of o as performed in a static setting over the *worst-case state*, during the lifetime of o, of G. It can be shown that the worst-case state of G that o can encounter is $\overline{S_{G,o}} = (O(n + \widetilde{I_o}), O(m + \widetilde{I_o}), O(\delta + \widetilde{I_o}))$.

▶ **Theorem 1.** *Denote* $\mathscr{Q}$=$\{BFS, SSSP, BC\}$, $\mathscr{M}_V$=$\{\textsc{Put}V, \textsc{Rem}V\}$, $\mathscr{M}_E$=$\{\textsc{Put}E, \textsc{Rem}E\}$, *and* $\mathscr{M} = \mathscr{M}_V \cup \mathscr{M}_E$. *Let* δ_e *be the degree of vertex whose edge modification happens. Denote* $X_{\mathscr{U}} = \{o \in X \mid type(o) \in \mathscr{U}, \mathscr{U} \subseteq \mathscr{A}\}$, *where* $\mathscr{A}$ *is the ADT as defined before. Let* $I_{o,\mathscr{U}}$ *and* $C_{o,\mathscr{U}}$ *denote the interval and point contentions, respectively, of* o *pertaining to the operation calls* $o \in \{X_{\mathscr{U}} \cup \{o\}\}$. *Accordingly,* $\widetilde{I_{o,\mathscr{U}}} = I_{o,\mathscr{U}} - 1$. *Considering the queries* $q \in \mathscr{Q}$ *performed by PG-Cn, the worst-case amortized cost per operation* A_X *for an execution* X *s.t.* $type(o) \in \mathscr{M} \cup \{q\}\ \forall o \in X$, *and* $q \in \mathscr{Q}$ *is* $A_X = A_X(\mathscr{M}) + \frac{C_{o,\mathscr{M}}}{|X|} \sum_{o \in X_{\mathscr{Q}}} \left(W_{o,\overline{S_{G,o}}} + \widetilde{I_{o,\mathscr{M}}}\right)$, *where* $A_X(\mathscr{M}) = \frac{C_{o,\mathscr{M}_V}}{|X|} \sum_{o \in X_{\mathscr{M}_V}} W_{o,\overline{S_{G,o}}} + \frac{1}{|X|} \sum_{o \in X_{\mathscr{M}_X}} W_{o,\overline{S_{G,o}}} + \frac{C_{o,\mathscr{M}_E}}{|X|} \sum_{o \in X_{\mathscr{M}_E}} O(\delta_e)$.

The proof of Theorem 1 is available in the full version [8].

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873–890, 1993.

2 Yehuda Afek, Gideon Stupp, and Dan Touitou. Long Lived Adaptive Splitter and Applications. *Distributed Comput.*, 15(2):67–86, 2002.

3 Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In *(PDP) 2019*, pages 83–92, 2019.

4 Hagit Attiya and Arie Fouren. Algorithms Adapting to Point Contention. *J. ACM*, 50(4):444–468, 2003.

5 Sumit Bhatia, Bapi Chatterjee, Deepak Nathani, and Manohar Kaul. A Persistent Homology Perspective to the Link Prediction Problem. In *Complex Networks*, page (to appear), 2019.

6 Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara, and Alessandro Provetti. Crawling Facebook for Social Network Analysis Purposes. In *(WIMS)*, page 52, 2011.

7 Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

8 Bapi Chatterjee, Sathya Peri, and Muktikanta Sa. Dynamic Graph Operations: A Consistent Non-blocking Approach. *CoRR*, abs/2003.01697, 2020.

9 Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

10 Antonio del Sol, Hirotomo Fujihashi, and Paul O'Meara. Topology of Small-world Networks of Protein–Protein Complex Structures. *Bioinformatics*, 21(8):1311–1315, 2005.

11 Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *40th PLDI*, pages 918–934, 2019.

12 D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. In *HPEC*, 2012.

13 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

14 Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving Graph Processing at Scale. In *GRADES*, page 5, 2016.

15 Wole Jaiyeoba and K. Skadron. Graphtinker: A high performance data structure for dynamic graph processing. *IPDPS*, pages 1030–1041, 2019.

16 P. Kumar and H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *FAST*, 2019.

17 Julian Shun and Guy E. Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*, pages 135–146, 2013.

18 Keval Vora, R. Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ASPLOS*, 2017.

Brief Announcement: Auditable Register Emulations

Vinicius Vielmo Cogo
LASIGE, Faculdade de Ciências, University of Lisbon, PT

Alysson Bessani
LASIGE, Faculdade de Ciências, University of Lisbon, PT

Abstract

We initiate the study of auditable storage emulations, which provide the capability for an auditor to report the previously executed reads in a register. We define the notion of `auditable register` and its properties, and establish tight bounds and impossibility results for auditable storage emulations in the presence of faulty base storage objects. Our formulation considers registers that securely store data using information dispersal (each base object stores only a block of the written value) and supporting fast reads (that complete in one communication round-trip). In such a scenario, given a maximum number f of faulty storage objects and a minimum number τ of data blocks required to recover a stored value, we prove that (R1) auditability is impossible if $\tau \leq 2f$; (R2) implementing a weak form of auditability requires $\tau \geq 3f + 1$; and (R3) a stronger form of auditability is impossible. We also show that (R4) signing read requests generically overcomes the lower bound of weak auditability, while (R5 and R6) totally ordering operations or using non-fast reads enables strong auditability. These results establish that practical storage emulations need f to $2f$ additional objects compared to their original lower bounds to support auditability.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Computer systems organization → Reliability; Security and privacy → Information accountability and usage control; Applied computing → Evidence collection, storage and analysis

Keywords and phrases Auditability, Secure Storage, Information Dispersal

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.53

Related Version *Full Version*: `https://arxiv.org/abs/1905.08637`

Funding Work partially supported by the Fundação para a Ciência e Tecnologia (FCT, Portugal), through the LASIGE research unit (UIDB/00408/2020 and UIDP/00408/2020) and the IRCoC project (PTDC/EEI-SCR/6970/2014).

1 Introduction

Given a resilient storage system composed of n storage objects (e.g., [2, 11]), *information dispersal* techniques (e.g., erasure codes and secret sharing) traditionally split and convert a data value v into n coded blocks [7, 9, 10]. Each coded block b_{v_k} from value v is stored in a different base object o_k, and readers need to obtain only τ out of n coded blocks to effectively recover the original value v. In this type of solution, no base object stores the whole data value, which differentiates information dispersal from fully-replicated storage systems where each object retains a full copy of the value.

In this paper, we address the following question: *How to extend resilient storage emulations with the capability of auditing who has effectively read data from them?* More specifically, we intend to audit resilient storage systems for protecting them from readers trying to obtain data without being detected (i.e., audit completeness) and protecting correct readers from faulty storage objects trying to incriminate them (i.e., audit accuracy).

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 53; pp. 53:1–53:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Model. We consider a system composed of an arbitrary number of *client processes* (writers, readers, and auditors) that interact with a set of *n base storage objects* by invoking operations (e.g., as remote procedure calls on top of asynchronous reliable, authenticated channels).

Faulty writers and *faulty auditors* are honest and can only fail by crashing. *Faulty readers* may be Byzantine, i.e., they can crash or request data from only a subset of objects, without complying with any read algorithm. This characterises an attack where they attempt to read data without being detected and reported by auditors.

Faulty storage objects can crash, omit their blocks to readers, omit read records to auditors, or record nonexistent read operations. More specifically, to isolate storage from logging, objects can fail by omission when accessing their blocks and arbitrarily when providing their log records. Omitting records to auditors means a faulty object may be helping a reader to avoid being detected by auditors. Producing records for nonexistent reads characterises an active attack where a faulty object may be trying to incriminate a reader. Furthermore, we assume no more than f storage objects are faulty.

Register Emulation. We consider a `dispersed register` as a *high-level shared storage object* that stores a value v using information dispersal schemes. This object provides two high-level operations: *a-write(v)* and *a-read()*. A high-level *a-write(v)* converts a value v, passed as an argument, into n coded blocks $b_{v_1}, b_{v_2}, ..., b_{v_n}$, and each coded block b_{v_k} is stored in the base object o_k. A high-level *a-read()* operation recovers the original value v from any subset consisting of a specific number τ of distinct blocks b_{v_k}.

Base objects in this work are `loggable R/W registers`, which is an object o_k that stores a data block b_{v_k} and has a log L_k to store records of every read operation that this base object responded to. This object o_k provides three low-level operations:

- *rw-write*(b_{v_k}): writes the data block b_{v_k}, passed as an argument, in this base object o_k and returns an *ack* to confirm that the operation succeeded.
- *rw-read()*: returns the data block b_{v_k} currently stored in this base object o_k (i.e., the block passed as argument in the latest preceding *rw-write*) or $\perp$ if no block has been written on it. It also creates a record $\langle p_r, \mathit{label}(b_{v_k})\rangle$[1] about this read in its log L_k.
- *rw-getLog()*: returns the log L_k of this base object o_k.

We extend the `dispersed register` emulation with a high-level operation *a-audit()*, which uses the fail-prone logs obtained from the *rw-getLog* operation in `loggable R/W registers` to compose an `auditable register` emulation. This emulation has access to a log $L \subseteq \bigcup_{k \in \{1..n\}} L_k$ from which auditors can infer who has effectively read a value from the register in the past. Four concepts are essential for our initial `auditable register` emulation: *providing sets*, *effective reads*, *fast reads*, and *available auditing quorums.*

First, we define a *providing set* based on the notion of an accepting set introduced by Lamport [8] to abstract the access to multiple base objects. In our work, a providing set $P_{p_r,v}$ is *maximal* if it contains all objects that have both stored a block associated with value v and returned this block to a reader p_r.

Second, we introduce the notion of an *effective read*, which characterises a providing set $P_{p_r,v}$ large enough (i.e., comprising at least τ base objects) that a reader p_r is able to effectively obtain value v from the received blocks.

[1] p_r is the identifier of the reader that invoked the *rw-read* operation and $\mathit{label}(b_{v_k})$ is an auxiliary function that, given a block b_{v_k}, returns a label (e.g., a unique identifier, hash, timestamp) associated to value v (from which the block b_{v_k} was derived). This label can be written in the first few bytes of each block and is critical for auditors to detect effective reads based on individual logs.

Third, to capture the most fundamental aspect of reading in dispersed storage, we consider (initially) that *high-level reads are fast* – i.e., each read completes in a single communication round-trip between the reader and the storage objects [4]. This faithfully represents a "stealthy" reader directly obtaining blocks from a number of storage objects without following any particular algorithm.

Fourth, we define an *available auditing quorum* A, where $|A| = n - f$, as the set of base objects from which the *a-audit* collects fail-prone individual logs to compose a *set of evidences* E_A about the effectively read values. Furthermore, we assume a threshold δ as the *minimal required number of collected records* obtained in the audit operation for an auditor create an evidence $\mathcal{E}_{p_r,v}$ of an effective read. Each *evidence* $\mathcal{E}_{p_r,v}$ contains at least δ records from different storage objects o_k proving that v was effectively read by reader p_r. This threshold δ is a configurable parameter that depends on the guarantees *a-audit* operations provide (defined below). A correct auditor receives E_A and reports all evidenced reads.

An `auditable register` provides an *a-audit* operation that guarantees *completeness* and at least one form of *accuracy* (i.e., weak or strong). The formal definitions of these properties are available in the full paper [3]. In summary, the completeness states that all effective reads that precede an *a-audit* are reported by auditors in this audit operation. The weak accuracy states that the *a-audit* operation never reports any effective read from a correct reader that has never tried to read any value. The strong accuracy states that the *a-audit* does not report any effective read of value v from a correct reader that has never effectively read this value v. In the remaining of this paper, we consider *weak auditability* when the storage system provides completeness and weak accuracy in audit operations and *strong auditability* when it provides completeness and strong accuracy.

2 Results

Several tight bounds and impossibility results are presented in the full paper [3] considering the previously mentioned model. We consider information dispersal as the primary form of `auditable register` emulations because alternative solutions that replicate the whole data can suffer from faulty base objects leaking data to readers without logging these read operations.

Our formulation stores data using information dispersal (each base object stores only a block of the written value) and initially supports fast reads (that complete in one communication round-trip). In such a scenario, given a maximum number f of faulty storage objects and a minimum number τ of data blocks required to recover a stored value, we prove that (R1) auditability is impossible if $\tau \leq 2f$; (R2) implementing a weak form of auditability requires $\tau \geq 3f + 1$; and (R3) a stronger form of auditability is impossible.

We also prove in [3] that (R4) signing read requests generically (i.e., without mentioning the value v to be read) overcomes the lower bound of weak auditability, (R5) totally ordering operations [5] or using non-fast reads (e.g., multi-round read algorithms [2]) enables strong auditability with $\tau \geq 3f + 1$, and (R6) combining non-fast reads with specific signed read requests (i.e., read requests with signatures valid only for a specific value v) overcomes the lower bound of strong auditability with $\tau \geq 2f + 1$. These results establish that practical storage emulations (e.g., [1, 2, 6]) need f to $2f$ additional objects compared to their original lower bounds to support auditability.

References

1 Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K. Reiter, and Emin Gün Sirer. Efficient verifiable secret sharing with share recovery in BFT protocols. In *Proc. of the 26th ACM Conference on Computer and Communications Security (CCS)*, page 2387–2402, 2019. doi:10.1145/3319535.3354207.

2 Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12:1–12:33, 2013. doi:10.1145/2535929.

3 Vinicius Vielmo Cogo and Alysson Bessani. Auditable register emulations. *CoRR*, abs/1905.08637, 2019. arXiv:1905.08637.

4 Rachid Guerraoui and Marko Vukolić. How fast can a very robust read be? In *Proc. of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 248–257, 2006. doi:10.1145/1146381.1146419.

5 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

6 James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 73–86, 2007. doi:10.1145/1294261.1294269.

7 Hugo Krawczyk. Secret sharing made short. In *Proc. of the 13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, pages 136–146, 1993. doi:10.1007/3-540-48329-2_12.

8 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006. doi:10.1007/s00446-006-0155-x.

9 James S Plank. Erasure codes for storage systems: A brief primer. *The USENIX Magazine*, 38(6):44–50, 2013.

10 Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM (JACM)*, 36(2):335–348, 1989. doi:10.1145/62044.62050.

11 Jason K. Resch and James S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proc. of the 9th USENIX Conference on File and Storage Technologies (FAST)*, page 14, 2011.

Brief Announcement: Accountability and Reconfiguration – Self-Healing Lattice Agreement

Luciano Freitas de Souza
CEA LIST, Université de Paris-Saclay, France

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Thibault Rieutord
CEA LIST, Université de Paris-Saclay, France

Sara Tucci-Piergiovanni
CEA LIST, Université de Paris-Saclay, France

Abstract

An *accountable* distributed system provides means to detect deviations of system components from their expected behavior. It is natural to complement fault detection with a reconfiguration mechanism, so that the system could heal itself, by replacing malfunctioning parts with new ones. In this paper, we describe a framework that can be used to implement a large class of accountable and reconfigurable replicated services. We build atop the fundamental lattice agreement abstraction lying at the core of storage systems and cryptocurrencies.

Our asynchronous implementation of accountable lattice agreement ensures that every violation of consistency is followed by an undeniable evidence of misbehavior of a faulty replica. The system can then be seamlessly reconfigured by evicting faulty replicas, adding new ones and merging inconsistent states. We believe that this paper opens a direction towards asynchronous "self-healing" systems that combine accountability and reconfiguration.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Reconfiguration, accountability, asynchronous, lattice agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.54

Related Version *Full Version*: `https://arxiv.org/abs/2105.04909`

1 Introduction

There are two major ways to deal with failures in distributed computing:

Fault-tolerance: we anticipate failures by investing into replication and synchronization, so that the system's correctness is not affected by faulty components.

Accountability: we detect failures *a posteriori* and raise undeniable evidences against faulty components.

Accountability in computing has been proposed for generic distributed systems [13, 14] as a mechanism to detect deviations of system nodes from the algorithms they are assigned with. It has been shown that a large class of deviations of a given process from a given deterministic algorithm can be detected by maintaining a set of *witnesses* that keep track of all *observable* actions of the process and check them against the algorithm [15].

The generic approach can be, however, very expensive in practice and one may look for a more tractable *application-specific* accountability mechanism. Indeed, instead of pursuing the ambitious goal of detecting deviations from the assigned algorithm, we might want to only care about deviations that violate the specification of the problem the algorithm is trying to solve.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 54; pp. 54:1–54:5
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Application-specific accountability. The idea has been successfully employed in the context of Byzantine Consensus [6]. The accountable version of consensus guarantees correctness as long as the number of faulty processes does not exceed some fixed f. But if correctness is violated, e.g., honest processes take different decisions, then at least $f+1$ Byzantine processes are presented with undeniable evidences of misbehavior. This is not surprising: a decision in a typical f-resilient consensus protocol must receive *acknowledgements* from a *quorum* of processes, and any two quorums must have at least $f+1$ processes in common [20]. The fact that two processes took different decisions implies that at least $f+1$ processes in the intersection of the corresponding quorums *equivocated*, i.e., acknowledged conflicting decision values. Assuming that every decision is provided with a cryptographic *certificate* containing the set of signed acknowledgements from a quorum of processes, we can immediately construct a desired evidence. Polygraph [6, 7], a recent accountable Byzantine Consensus protocol, naturally builds upon the classical PBFT protocol [5]. One may ask – okay, we have detected a faulty process, but what should we do next? Ideally, we would like to *reconfigure* the system by evicting the faulty process and *reinitializing* the system state.

Reconfigurable replicated systems [11, 12, 16, 22] allow the users to dynamically update the set of replicas. It has been recently shown that reconfiguration can be implemented in purely *asynchronous* environments [1, 2, 11, 16, 17, 22]. The idea was first applied to (read-write) storage systems [1, 2, 11], and then extended to max-registers [16, 22] and more general *lattice* data types, first in the crash-fault context [17] and then for Byzantine failures [18].

Contribution. In this paper, we propose a framework that can be used to implement a large class of replicated services that are both accountable and reconfigurable. Following recent work on reconfiguration [16–18], we build atop the fundamental *lattice agreement* abstraction. Lattice agreement [3, 9] takes arbitrary inputs in a *lattice* (a partially ordered set equipped with a *join* operator) and returns outputs that are (1) joins of the inputs, and (2) ordered with respect to the lattice partial order. Lattice agreement is weaker than consensus and can be implemented in an asynchronous system.

Lattice agreement (LA) appears to be a perfect match for both desired features: accountability and reconfiguration. Indeed, a quorum-based LA implementation enables detection of misbehaving parties: as soon as two correct users learn two incomparable values, they also obtain a proof of misbehavior of all replicas that *signed* both values. Furthermore, the very process of reconfiguration can be represented as agreement defined on a lattice of *configurations* [16, 17]. These two observations inspire the design of our system.

We propose an accountable *and* reconfigurable implementation that reaches agreement on a *joint* lattice: an object lattice (defining the current *state* of the replicated object) and a configuration lattice (defining the current *configuration* of the replicas). Assuming that the number of failures is less than half of the system size, our implementation is *alive*. It is also *safe* if only benign (crash) failures occur. Once safety is violated, i.e., two correct users learn two incomparable object states, some Byzantine replicas are inevitably confronted with an undeniable proof of misbehavior. The system is then seamlessly reconfigured by evicting the detected replicas, adding new ones and merging inconsistent states. Once the state is merged, the system comes back to providing safety and liveness, as long as no new replicas exhibits Byzantine behavior. Eventually all Byzantine replicas are detected and the system maintains liveness and safety.

Outdated configurations are harmless. Our system prevents users from accessing outdated configurations with the use of *forward-secure digital signature scheme* [4,8]. A member of each new configuration is assigned a new secret key. Furthermore, honest members of an old configuration are expected to destroy their old keys before moving to a new one. Thus, if they are later compromised, they will not be able to serve clients' requests, and the remaining Byzantine replicas will not constitute a quorum.

On Byzantine clients. Our solution assumes that service replicas are subject to Byzantine failures, but clients are *benign*: they can only fail by crashing. This hypothesis has already been done in designs of fault-tolerant storage systems [19]. In our case, this assumption precludes the cases when a Byzantine client brings the system into a compromised configuration or slows down the system by issuing excessive reconfiguration requests. In the full version of this paper [10], we also describe a *one-shot* version of accountable lattice agreement, without reconfiguration, in which *both* clients and replicas can be Byzantine. Marrying reconfiguration and accountability in a long-lived service that can be accessed by Byzantine clients remains an important challenge.

Message. Altogether, we believe that this paper opens a new area of asynchronous "self-healing" systems that combine accountability and reconfiguration. Such a system either preserves safety and liveness or preserves liveness and compensates safety violations with eventual detection of Byzantine replicas. It also exports a reconfiguration interface that allows the clients to replace compromised replicas with new, correct ones. In this paper, we show that both mechanisms, accountability and reconfiguration, can be implemented in a purely asynchronous (in the modern parlance – *responsive*) way.

2 Reconfigurable and Accountable Lattice Agreement

A *reconfigurable accountable (long-lived) lattice agreement* (*RALA*) abstraction must ensure, among others, the following properties:

- **Completeness.** If a correct client learns a value that is incomparable with a value learnt by another correct client then it eventually accuses some new replicas.
- **Liveness.** If the system reconfigures only finitely many times, every value proposed by a correct client is eventually included in the value learned by every correct client.

The properties above imply that either the safety property of the implemented object holds (the values learnt by correct processes are comparable) or some new Byzantine replicas are eventually detected. If from some point on, no more Byzantine faults take place, we ensure that all new learnt values are comparable. Our requirement of finite number of reconfigurations is standard in the corresponding literature [2,17,22] and, in fact, can be shown to be necessary [21]. In practice, we ensure liveness in "sufficiently long" time intervals without reconfiguration.

Notice that the choice of new configurations to propose is left entirely to the clients, as long as one condition is satisfied: if the system is in a configuration which is not eventually replaced, than this configuration must contain a majority of correct replicas. It is important to emphasize that **the system does not allow the accused replicas to affect the system's safety and liveness**. Please refer to the full version [10] for the complete specification and the matching implementation.

References

1 Marcos Kawazoe Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, 2011.

2 Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni da Silva Fraga. Efficient and modular consensus-free reconfiguration for fault-tolerant storage. In *OPODIS*, pages 26:1–26:17, 2017.

3 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Comput.*, 8(3):121–132, 1995.

4 Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.

5 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

6 Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. *IACR Cryptol. ePrint Arch.*, 2019:587, 2019. URL: https://eprint.iacr.org/2019/587.

7 Pierre Civit, Seth Gilbert, and Vincent Gramoli. Brief announcement: Polygraph: Accountable byzantine agreement. In Hagit Attiya, editor, *DISC*, volume 179 of *LIPIcs*, pages 45:1–45:3, 2020.

8 Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers.

9 Jose Faleiro, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, pages 125–134, 2012.

10 Luciano Freitas de Souza, Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Accountability and reconfiguration: Self-healing lattice agreement. *CoRR*, abs/2105.04909, 2021. arXiv:2105.04909.

11 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *DISC*, pages 140–153, 2015.

12 Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo: a robust, reconfigurable atomic memory service for dynamic networks. *Distributed Comput.*, 23(4):225–272, 2010.

13 Andreas Haeberlen and Petr Kuznetsov. The Fault Detection Problem. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS'09)*, December 2009.

14 Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. The case for byzantine fault detection. In *Proceedings of the Second Workshop on Hot Topics in System Dependability (HotDep'06)*, November 2006.

15 Andreas Haeberlen, Petr Kuznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

16 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *DISC*, pages 154–169, 2015.

17 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci-Piergiovanni. Reconfigurable lattice agreement and applications. In *OPODIS*, 2019.

18 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In Hagit Attiya, editor, *DISC*, volume 179 of *LIPIcs*, pages 27:1–27:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

19 Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In Dahlia Malkhi, editor, *DISC*, volume 2508 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2002.

20 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

21 Alexander Spiegelman and Idit Keidar. On liveness of dynamic storage. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO 2017, Porquerolles, France, June 19-22, 2017, Revised Selected Papers*, pages 356–376, 2017.

22 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.

Brief Announcement: On Strong Observational Refinement and Forward Simulation

John Derrick
University of Sheffield, UK

Simon Doherty
University of Sheffield, UK

Brijesh Dongol
University of Surrey, UK

Gerhard Schellhorn
Universität Augsburg, Germany

Heike Wehrheim
Universität Oldenburg, Germany

Abstract

Hyperproperties are correctness conditions for labelled transition systems that are more expressive than traditional trace properties, with particular relevance to security. Recently, Attiya and Enea studied a notion of strong observational refinement that preserves all hyperproperties. They analyse the correspondence between forward simulation and strong observational refinement in a setting with finite traces only. We study this correspondence in a setting with both finite and infinite traces. In particular, we show that forward simulation does not preserve hyperliveness properties in this setting. We extend the forward simulation proof obligation with a progress condition, and prove that this *progressive forward simulation* does imply strong observational refinement.

2012 ACM Subject Classification Security and privacy → Formal methods and theory of security; Computing methodologies → Concurrent computing methodologies

Keywords and phrases Strong Observational Refinement, Hyperproperties, Forward Simulation

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.55

Related Version *Full Version*: `https://arxiv.org/abs/2107.14509`

Funding *John Derrick*: EPSRC Grant EP/R032351/1.
Brijesh Dongol: EPSRC grants EP/V038915/1, EP/R032556/1, EP/R025134/2 and VeTSS.
Heike Wehrheim: DFG Grant SFB 901.

1 Introduction

Hyperproperties [2] form a large class of properties over sets of sets of traces, characterising, in particular, security properties such as generalised non-interference that are not expressible over a single trace. Like with trace properties, every hyperproperty can be characterised as the conjunction of a hypersafety and hyperliveness property.

Recently, Attiya and Enea proposed *strong observational refinement*, a correctness condition that preserves all hyperproperties, even in the presence of an adversarial scheduler. An object O_1 strongly observationally refines an object O_2 if the executions of any program P using O_1 as scheduled by some admissible deterministic scheduler cannot be observationally distinguished from those of P using O_2 under another deterministic scheduler. They showed that strong observational refinement preserves all hyperproperties. Furthermore, they prove that forward simulation implies strong observational refinement. Forward simulation alone is sound but not complete for ordinary refinement, and in general both backward and forward

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 55; pp. 55:1–55:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

simulation are required. Forward simulation is furthermore known to not preserve liveness properties, which motivates our study of forward simulation and observational refinement in the context of infinite traces and hyperliveness.

As a result we show – by example – that forward simulation does not preserve hyperliveness. Furthermore, forward simulation alone cannot guarantee strong observational refinement when requiring admissiblity of schedulers, i.e., when schedulers are required to continually schedule enabled actions. To address these limitations, we employ a version of forward simulation extended with a progress condition, thereby guaranteeing strong observational refinement and preservation of hyperliveness.

2 Motivating Example

```
int* current_val initially 0
int fetch_and_add(int k):
F1. do n = LL(&current_val)
F2. while (!SC(&current_val, n + k))
F3. return n
```

Figure 1 A fetch-and-add with a nonterminating schedule when LL and SC are implemented using the algorithm of [4].

We give an example of an abstract atomic object O_2 and a non-atomic implementation O_1 such that there *is* a forward simulation from O_1 to O_2, but hyperliveness properties are not preserved for all schedules. As the atomic abstract object O_2 we choose a *fetch-and-add* object with just one operation, `fetch_and_add(int k)`, which adds the value integer `k` to a shared integer variable and returns the value of that variable before the addition. Let P be a program with two threads t_1 and t_2, each of which executes one `fetch_and_add` operation and assigns the return value to a local variable of the thread. For any scheduler S, the variable assignment of both threads will eventually occur. This "eventually" property can be expressed as a hyperproperty.

Now, consider the *fetch-and-add* implementation presented in Figure 1. This implementation uses the `load-linked/store-conditional` (LL/SC) instruction pair. The `LL(ptr)` operation loads the value at the location pointed to by the pointer `ptr`. The `SC(ptr,v)` conditionally stores the value `v` at the location pointed to by `ptr` if the location has not been modified by another `SC` since the executing thread's most recent `LL(ptr)` operation. If the update actually occurs, `SC` returns `true`, otherwise the location is not modified and `SC` returns `false`. In the first case, we say that the `SC` *succeeds*. Otherwise, we say that it *fails*.

Critically, we stipulate that the LL and SC operations are implemented using the algorithm of [4]. This algorithm has the following property. If thread t_1 executes an `LL` operation, and then thread t_2 executes an `LL` operation *before* t_1 has executed its subsequent `SC` operation, then that `SC` is guaranteed to fail. This happens even though there is no intervening modification of the location.

Now, let O_1 be a *labelled transition system* (LTS) representing a multithreaded version of this `fetch_and_add` implementation, using the specified `LL/SC` algorithm. Consider furthermore the program P (above) running against the object O_1. A scheduler can continually alternate the `LL` at line `F2` of t_1 and that of t_2, such that neither `fetch_and_add` operation ever completes. Therefore, unlike when using the O_2 object, the variable assignments of P will never occur, so the O_1 system does not satisfy the hyperproperty for all schedulers.

There is, however, a forward simulation from O_1 to O_2. The underlying `LL`/`SC` implementation can be proven correct by means of forward simulation, as can the `fetch_and_add` implementation. Therefore, standard forward simulation is insufficient to show that all hyperproperties are preserved, contradicting Lemma 5.2 of [1].

3 Progressive Forward Simulation implies Strong Observ. Refinement

We will use the notation of Attiya and Enea [1], in particular that of an LTS $A = (Q, \Sigma, s_0, \delta)$ and of a (deterministic, admissible) scheduler $S : \Sigma^* \rightarrow 2^\Sigma$ (full definitions can also be found in [3]). The main change we make is that the traces in trace sets $T(A)$ and $T(A, S)$ (S-scheduled traces) now include finite and infinite sequences[1]. A scheduler is *admitted* by an LTS A if for all finite traces σ of A consistent with S, the scheduler satisfies (i) $S(\sigma)$ is non-empty and (ii) all actions in $S(\sigma)$ are enabled in *state*(σ). Besides being admissible, schedulers for programs P and objects O (LTSs of the form $P \times O$) also have to be *deterministic*: they must resolve the nondeterminism on the actions of the object. An object O_1 *strongly observationally refines* the object O_2, written $O_1 \leq_S O_2$, iff for every deterministic scheduler S_1 admitted by $P \times O_1$ there exists a deterministic scheduler S_2 admitted by $P \times O_2$ such that $T(P \times O_1, S_1)|\Sigma_P = T(P \times O_2, S_2)|\Sigma_P$ for all programs P.

Contrary to the claim in [1], standard forward simulation does not imply strong observational refinement (details in [3]). In the example given in Section 2, a deterministic admissible scheduler S_1 for P and O_1 could drive $P \times O_1$'s execution along the infinite trace of `LL` and `SC` operations, so that calls to `fetch_and_add` never return. On the other hand, any scheduler for the O_2 system must eventually execute call and return actions for both `fetch_and_add` operations, and subsequently execute the writes to the program variables. Thus, $T(P \times O_1, S_1)|\Sigma_P \neq T(P \times O_2, S_2)|\Sigma_P$. To guarantee strong observational refinement, forward simulation additionally has to guarantee some sort of progress, so that the scheduler S_2 is always able to schedule some action without producing a trace not present in $P \times O_1$ under S_1. This guarantee can be made if we disallow infinite stuttering.

▶ **Definition 1** (Progressive Forward Simulation). *Let $A_i = (Q_i, \Sigma_i, s_0^i, \delta_i)$, $i = 1, 2$, be two LTSs and Γ an alphabet. A relation $F \subseteq Q_1 \times Q_2$ together with a well-founded order $\ll \subseteq Q_1 \times Q_1$ is called a* progressive Γ-forward simulation *from A_1 to A_2 iff*

- $(s_0^1, s_0^2) \in F$, *and*
- *for all $(s_1, s_2) \in F$, if $(s_1, a, s_1') \in \delta_1$ and $a \in \Sigma_1$, then there exist $\alpha \in \Sigma_2^*$ and $s_2' \in Q_2$ such that $a \mid \Gamma = \alpha \mid \Gamma$, $(s_2, \alpha, s_2') \in \delta_2$ and $(s_1', s_2') \in F$. Whenever $\alpha = \varepsilon$ then $s_1' \ll s_1$.*

The definition requires that the concrete state decreases in the well-founded order when the abstract sequence α in the forward simulation is empty and $s_2 = s_2'$ (stuttering). Progressiveness prohibits an infinite sequence of concrete internal steps that map to the empty abstract sequence. For object O_1 above with the `fetch_and_add` implementation no such well-founded ordering can be given. We have (full proof in [3]):

▶ **Theorem 2.** *If there exists a progressive $(C \cup R)$-forward simulation from O_1 to O_2, then $O_1 \leq_S O_2$.*

[1] The work of [1] just considers finite traces. However, they still assume schedulers to always be able to schedule a next action which seems to contradict the fact that all traces are finite.

4 Conclusion

In this paper, we have reported on our findings that forward simulation does not imply strong observational refinement in a setting with infinite traces. We have proposed a notion of progressive forward simulation implying strong observational refinement. In future work, we will investigate whether the reverse direction also holds.

References

1 H. Attiya and C. Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In J. Suomela, editor, *DISC*, volume 146 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl, 2019. doi:10.4230/LIPIcs.DISC.2019.2.

2 M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

3 J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. On strong observational refinement and forward simulation, 2021. arXiv:2107.14509.

4 V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *SPAA*, pages 314–323. ACM, 2003. doi:10.1145/777412.777468.

Brief Announcement: Persistent Software Combining

Panagiota Fatourou
Foundation for Research and Technology – Hellas, Institute of Computer Science, Heraklion, Greece
University of Crete, Department of Computer Science, Heraklion, Greece

Nikolaos D. Kallimanis
Foundation for Research and Technology – Hellas, Institute of Computer Science, Heraklion, Greece

Eleftherios Kosmas
University of Crete, Department of Computer Science, Heraklion, Greece

Abstract

We study the performance power of software combining in designing recoverable algorithms and data structures. We present two recoverable synchronization protocols, one blocking and another wait-free, which illustrate how to use software combining to achieve both low persistence and synchronization cost. Our experiments show that these protocols outperform by far state-of-the-art recoverable universal constructions and transactional memory systems. We built recoverable queues and stacks, based on these protocols, that exhibit much better performance than previous such implementations.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Data structures design and analysis

Keywords and phrases Persistent objects, recoverable algorithms, durability, synchronization protocols, software combining, universal constructions, wait-freedom, stacks, queues

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.56

Related Version *Full Version*: `https://arxiv.org/abs/2107.03492`

Funding *Panagiota Fatourou*: Supported by the EU Horizon 2020, Marie Sklodowska-Curie project with GA No 101031688. *Eleftherios Kosmas:* Co-financed by Greece and the European Union (European Social Fund- ESF) through the Operational Programme «Human Resources Development, Education and Lifelong Learning» in the context of the project "Reinforcement of Postdoctoral Researchers - 2nd Cycle" (MIS-5033021), implemented by the State Scholarships Foundation (IKY).

1 Motivation and Contribution

The availability of byte-addressable non-volatile main memory (NVMM) enables the design of recoverable concurrent algorithms. An algorithm is recoverable (also known as *persistent*) if its state can be restored after recovery from a *system-crash* failure. Another important property, known as *detectability* [15], is to be able to determine, upon recovery, if a request has been completed, and if yes, to find its response.

When designing recoverable algorithms, the main challenge stems from the fact that all data stored into registers and caches are volatile. Thus, unless they have been flushed to persistent memory, such data will be lost at a system crash. Flushing to persistent memory occurs by including specific *persistence instructions*, such as `pwb`, `pfence` and `psync` in the code, which are however expensive in terms of performance. Despite many efforts for designing efficient recoverable synchronization protocols and data structures, persistence comes at an important cost even for fundamental data structures, such as queues and stacks.

We provide recoverable implementations of software combining protocols that exhibit much better performance and lower persistence cost, in comparison to a large collection of existing persistent techniques [3, 4, 22, 23] for achieving scalable synchronization. The implementation

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 56; pp. 56:1–56:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of many synchronization protocols, including queue-lock implementations [5, 19, 20] and software combining protocols [8, 9, 10, 11, 16, 21], require careful design to maintain them simple and low-cost; even minor changes in their code could negatively impact performance, rendering them uninteresting. This was also the case with our protocols. Coming up with them was quite intricate and they required careful design to ensure good performance.

As a recoverable implementation often inherents the synchronization overheads of the concurrent implementation it is inspired from, persistence should not be added on top of heavy universal constructions or transactional memory (TM) systems with high synchronization cost. To achieve this goal, we first design a new combining protocol, called BCOMB, that outperforms, by far, not only previous state-of-the art conventional combining protocols [8, 9, 10, 16], but also NUMA-aware synchronization techniques [6, 9]. Maintaining the number of persistence instructions low is also very important in terms of performance. We present PBCOMB, a recoverable blocking combining protocol built upon BCOMB, that reveals the power of appropriately utilizing software combining in achieving very low peristence cost.

We also present a wait-free recoverable universal construction, called PWFCOMB. PWFCOMB combines and extends ideas from PBCOMB and PSIM [8, 10]. PSIM [8, 10] is a state-of-the-art wait-free universal construction [7, 12, 13], which has been designed as the practical version of Herlihy's universal construction [17]. Our experimental analysis shows that both PBCOMB and PWFCOMB outperform by far their competitors [3, 4, 22, 23].

2 PBCOMB and PWFCOMB

Brief Description. PBCOMB follows the general idea of blocking software combining [9, 16, 21]. Each thread first announces its request and then tries to become the combiner. PBCOMB uses an array for storing the announced requests and a lock to identify which thread will become the combiner. After acquiring the lock, the combiner creates a copy of the state of the simulated object and applies the active requests on this copy. Then, the combiner switches a shared variable to index this copy, indicating that it stores the current valid state of the simulated object, and releases the lock. As long as the combiner serves active requests, other active threads perform local spinning, waiting for the combiner to release the lock.

PBCOMB owes its low persistence cost to the following two achievements: i) threads other than the combiner do not have to execute any persistence instruction, and ii) the combiner does not persist each of the requests it applies separately; it persists all the variables it access together, after it is done with the simulation of all requests it serves. Let the *combining degree* d be the average number of requests that a combiner serves. PBCOMB executes a small number of `pwb` instructions for every d requests. Achieving this, on top of a light-weight synchronization protocol, are the main reasons that PBCOMB has very good performance.

In PWFCOMB, many threads may simultaneously attempt to be the combiner to achieve wait-freedom: they copy the state of the object locally and use this local copy to apply all active requests they see announced. Then, each of them attempts to change a pointer S to point to its own local copy using SC. Only a single thread among them (the *combiner*) manages to do so. Achieving this form of wait-freedom using naive persistence approaches, would result in significant persistence overhead, as several threads attempt to become the combiner, and thus they all have to persist certain variables to ensure durability without blocking each other. Thus, we came up with more intricate persistence schemes to reduce this cost. PWFCOMB outperforms by far all competitors. However, it has worse performance than PBCOMB which pays less persistence overhead and is more light-weight.

Experimental Analysis. We performed our experimental analysis in a 48-core machine (96 logical cores) consisting of 2 Intel Xeon Platinum 8260M processors, each of which consists of 24 cores. The machine is equipped with a 1TB Intel Optane DC persistent memory

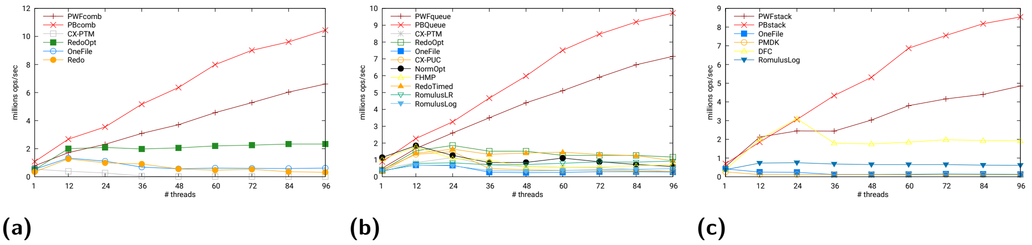


Figure 1 a) Average throughput of implementations while simulating a recoverable `AtomicFloat` object. b)-c) Average throughput of recoverable queue and stack implementations, respectively.

(DCPMM) (configured in AppDirect mode). Figure 1a shows that both, PBcomb and PWFcomb, outperform by far, many previous recoverable TM systems. Specifically, we compare the performance of PBcomb and PWFcomb against the following algorithms: OneFile [23], CX-PUC [4], CX-PTM [4], and RedoOpt [4]. For our experiments, we used the latest version of code for these algorithms that is provided in [2]. The diagrams show that PBcomb is 4x faster and PWFcomb is 2.8x faster than the competitors. Our protocols satisfy *detectability* [15] and are wait-free, whereas most competitors guarantee only weaker consistency (such as *durable linearizability* [18]), and/or progress.

Figures 1b and 1c illustrate that the recoverable queues (PBqueue and PWFqueue) and stacks (PBstack and PWFstack) that are built on top of PBcomb and PWFcomb, have much better performance than state-of-the-art recoverable implementations of such data structures, including the specialized recoverable queue implementations in [15]. Specifically, Figure 1b compares PBqueue and PWFqueue with the specialized recoverable queue implementation (FHMP) by Friedman *et al.* [15], the recoverable queue implementation (NormOpt) based on Capsules-Normal [1], and recoverable queue implementations based on Romulus [3] (RomulusLR and RomulusLog). Figure 1b shows that PWFqueue and PBqueue achieve superior performance, with PBqueue being more than 5x faster than the queue based on RedoOpt, which is the best competitor. Finally, Figure 1c compares the performance of PBstack and PWFstack against recoverable stack implementations based on OneFile [23] and Romulus [3], and against the archived recoverable stack based on flat-combining (DFC) [24]. PWFstack and PBstack exhibit much better performance than all the competitors. Specifically, PBstack is 4.4x faster than the DFC recoverable stack, which is the best competitor. See [14] for a full version of our paper which provides more diagrams and justification for the good performance of our algorithms.

3 Conclusion

The contributions of this paper can be summarized as follows.

- We present highly efficient recoverable software combining protocols that outperform *by far* many state-of-the-art recoverable universal constructions and TM systems.
- We built recoverable queues and stacks, based on our software combining protocols, which significantly outperform previous recoverable implementations of stacks and queues.
- Our results reveal the power of software combining in designing low cost persistent synchronization protocols and concurrent data structures.

References

1 N. Ben-David, G. E. Blelloch, M. Friedman, and Y. Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

2 A. Correia, P. Felber, and P. Ramalhete. The Code for RedoDB. URL: `https://github.com/pramalhe/RedoDB`.

3 A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, 2018.

4 A. Correia, P. Felber, and P. Ramalhete. Persistent memory and the rise of universal constructions. In *European Conference on Computer Systems (EuroSys)*, 2020.

5 T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Dept. of Computer Science, University of Washington, 1993.

6 D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. *ACM SIGPLAN Notices*, 47(8):247–256, 2012.

7 P. Fatourou and N. D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *International Symp. on Distributed Computing (DISC)*, pages 127–141, 2009.

8 P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

9 P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–626, 2012.

10 P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, 2014.

11 P. Fatourou and N. D. Kallimanis. Lock oscillation: Boosting the performance of concurrent data structures. In *Conf. on Principles of Distributed Systems (OPODIS)*, 2018.

12 P. Fatourou and N. D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33:485–513, 2020.

13 P. Fatourou, N. D. Kallimanis, and E. Kanellou. An efficient universal construction for large objects. In *Conf. on Principles of Distributed Systems (OPODIS)*, pages 18:1–18:15, 2018.

14 P. Fatourou, N. D. Kallimanis, and E. Kosmas. Persistent software combining. *CoRR*, abs/2107.03492, 2021. `arXiv:2107.03492`.

15 M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices*, 53(1):28–40, 2018.

16 D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symp. on Parallelism in algorithms and architectures (SPAA)*, pages 355–364, 2010.

17 M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

18 J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symp. on Distributed Computing (DISC)*, pages 313–327, 2016.

19 P. S. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *International Parallel Processing Symposium*, pages 165–171, 1994.

20 J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems (TOCS)*, 9(1):21–65, 1991.

21 Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, 1999.

22 PMDK. The persistent memory development kit. URL: `https://github.com/pramalhe/RedoDB`.

23 P. Ramalhete, A. Correia, P. Felber, and N. Cohen. Onefile: A wait-free persistent transactional memory. In *IEEE Conf. on Dependable Systems and Networks (DSN)*, pages 151–163, 2019.

24 M. Rusanovsky, O. Ben-Baruch, D. Hendler, and P. Ramalhete. A flat-combining-based persistent stack for non-volatile memory. *CoRR*, abs/2012.12868, 2020 (version submited at 23 December, 2020). `arXiv:2012.12868`.

Brief Announcement: Probabilistic Indistinguishability and The Quality of Validity in Byzantine Agreement

Guy Goren

The Viterbi Faculty of Electrical & Computer Engineering, Technion, Haifa, Israel

Yoram Moses

The Viterbi Faculty of Electrical & Computer Engineering, Technion, Haifa, Israel

Alexander Spiegelman

Novi Research, USA

Abstract

Lower bounds and impossibility results in distributed computing are both intellectually challenging and practically important. Hundreds if not thousands of proofs appear in the literature, but surprisingly, the vast majority of them apply to deterministic algorithms only. Probabilistic protocols have been around for at least four decades and are receiving a lot of attention with the emergence of blockchain systems. Nonetheless, we are aware of only a handful of randomized lower bounds.

In this work we provide a formal framework for reasoning about randomized distributed algorithms. We generalize the notion of indistinguishability, the most useful tool in deterministic lower bounds, to apply to a probabilistic setting. We apply this framework to prove a result of independent interest. Namely, we completely characterize the quality of decisions that protocols for a randomized multi-valued Consensus problem can guarantee in an asynchronous environment with Byzantine faults. We use the new notion to prove a lower bound on the guaranteed probability that honest parties will not decide on a possibly bogus value proposed by a malicious party. Finally, we show that the bound is tight by providing a protocol that matches it.

This brief announcement consists of an introduction to the full paper [6] by the same title. The interested reader is advised to consult the full paper for a detailed exposition.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security

Keywords and phrases Indistinguishability, probabilistic lower bounds, Byzantine agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.57

Related Version *Full Version*: https://arxiv.org/abs/2011.04719 [6]

Funding *Guy Goren*: Supported by a grant from the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau.
Yoram Moses: This work was funded in part by ISF grant 2061/19. Yoram Moses is the Israel Pollak academic chair at the Technion.

1 Introduction

Randomized algorithms have a long tradition in distributed computing [10], where they have been applied to many different problems in a variety of models [8]. In the context of fault-tolerant agreement they have served to overcome the impossibility of agreement in asynchronous settings [5, 11, 2], and have significantly improved efficiency compared to deterministic solutions [3, 7]. With the recent prevalence of blockchain systems, Byzantine agreement algorithms that can overcome malicious parties have found renewed interest in both industry and academia. For obvious reasons, blockchain systems should strive to

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 57; pp. 57:1–57:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

minimize the share of decisions that originate from malicious parties, and to increase the share originating from honest ones. A natural question, then, is what are the inherent limits on the quality of Byzantine agreement algorithms in this regard? Namely, what can we say about the probability with which an algorithm can guarantee that a good decision is made?

Given their practical importance, characterizing the power and limitations of randomized distributed algorithms for agreement has become ever more desirable. However, obtaining tight, or nontrivial, probabilistic bounds on properties in the asynchronous Byzantine setting can be a challenging task. As is well known, there are "*Hundreds of impossibility results for distributed computing*" [4]. But very few of them apply to randomized protocols. Unfortunately, there is currently a dearth of general tools for characterizing the properties of randomized algorithms.

The notion of indistinguishability has for years been one of the most useful tools for proving deterministic lower bounds and impossibility results in distributed computing [1]. Such deterministic lower bounds typically rely on the fact that if a correct party cannot distinguish between two executions of a deterministic protocol (i.e., its local state is the same in both), then it performs the same actions in both. In a randomized algorithm, the fact that two executions are indistinguishable to a given party up to a certain time does not ensure that its next action will be the same in both. Moreover, a single execution does not, by itself, provide information on the probability with which actions are performed. As a result, the classic notion of indistinguishability does not directly capture many of the probabilistic aspects of a randomized algorithm.

Of course, probabilistic properties of distributed algorithms such as "*the probability that the parties decide on a value proposed by an honest party is at least x*" or "*all honest parties terminate with probability 1*" cannot be evaluated based on an individual execution. Clearly, to make formal sense of such statements, we need to define an appropriate probability space. However, due the nondeterminism inherent in our model, a probability space over the set of all executions cannot be defined (cf. [9]). This is because we can't assume a distribution over the initial configurations, and similarly there is no well-defined distribution on the actions of the adversary, who is in charge of all the nondeterministic decisions. Once we fix the adversary's strategy, we are left with a purely probabilistic structure, which we call an *ensemble*. An ensemble naturally induces a probability space. This allows us to formally state probabilistic properties of an algorithm $\mathcal{A}$ of interest with respect to all of its ensembles (= adversary strategies). E.g., *"for every ensemble of algorithm $\mathcal{A}$, all honest parties terminate with probability 1."*

In deterministic algorithms, indistinguishability among executions is determined based on a party p_i's local history, i.e., the sequence of local states that p_i passes through in the executions. We generalize the notion of an i-local history to a notion called an *i-local ensemble*. A local ensemble is a tree of local states that captures subtle, albeit essential, aspects of probabilistic protocols. This facilitates the definition of a notion of **probabilistic indistinguishability** among ensembles, whereby two ensembles are considered indistinguishable to a process p_i if they induce identical i-local ensembles. Indistinguishability among ensembles provides a formal and convenient framework that can be used to simplify existing lower bound proofs in a probabilistic setting, and to prove new ones. A significant feature of this framework is its simplicity and ease of use, allowing similar arguments as in the deterministic case. The notions contain just enough structure beyond that of their deterministic analogues to capture the desired probabilistic properties.

Our original motivation for developing the above framework was to formally prove tight probabilistic bounds on the share of good decisions made by a randomized Byzantine agreement algorithm in an asynchronous setting. In Section 5 of [6] we use probabilistic indistinguishability to prove that, roughly speaking, no algorithm can guarantee that the probability to decide on a genuine input value is greater than $1 - \frac{f}{n-t}$. (As usual, n is the total number of parties here, while t and f are the maximal and actual number of failures, respectively.) Moreover, this bound is shown to be tight, by presenting an algorithm that achieves it.

The paper makes two distinct and complementary main contributions:

- We define a notion of indistinguishability that generalizes its deterministic counterpart, and is suitable for proving lower bounds in the context of probabilistic protocols. A new element in our definition is a purely probabilistic tree whose paths represent local histories of a given process. The resulting framework provides an intuitive and rigorous way to reason about probabilistic properties of such protocols.
- We introduce *Qualitative Validity*, a new probabilistic validity condition for the Byzantine agreement problem. It provides a probabilistic bound on the ability of corrupt parties to bias the decision values, which is of interest in the blockchain arena. We prove that, in a precise sense, it is the strongest achievable validity property in the asynchronous setting. Both the statement of the property and the proof are facilitated by our new framework.

We now provide an informal description of the Qualitative Validity condition, and present the lower bound and upper bound results regarding this condition that appear in the full version [6]. Roughly speaking, we use $\boldsymbol{f}$ to denote the maximal number of failures that a given adversary's strategy allows to occur in any of its possible executions. In addition, $max\_mult(\mathcal{V}_{in})$ denotes the multiplicity of the most frequent value in the input vector (consisting of the initial values of the n parties).

▶ **Definition 1** (Qualitative Validity). *If $max\_mult(\mathcal{V}_{in}) - \boldsymbol{f} \geq 2t + 1$, then all honest parties that decide, output decision values in $\mathcal{V}_{in}$. Otherwise, the probability that they decide on a value in $\mathcal{V}_{in}$ is at least $1 - \frac{\boldsymbol{f}}{n-t}$.*

Our main lower bound, which we prove using the probabilistic indistinguishability formalism introduced in this work, is stated as follows:

▶ **Theorem 2.** *No asynchronous Byzantine Agreement algorithm satisfies a validity property Φ that is strictly stronger than Qualitative Validity even against a weak and static adversary.*

In order to show that the lower bound result of Theorem 2 is tight, we present a Byzantine agreement protocol that satisfies Qualitative Validity. As a result we obtain:

▶ **Theorem 3.** *There exists an asynchronous Byzantine agreement algorithm that satisfies Qualitative Validity against a strong and adaptive adversary.*

References

1 Hagit Attiya and Faith Ellen. Impossibility results for distributed computing. *Synthesis Lectures on Distributed Computing Theory*, 5(1):1–162, 2014.

2 Michael Ben-Or. Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.

3 Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

4 Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed computing*, 16(2-3):121–163, 2003.

5 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 1985.

6 Guy Goren, Yoram Moses, and Alexander Spiegelman. Probabilistic indistinguishability and the quality of validity in byzantine agreement, 2020. arXiv:2011.04719.

7 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Annual International Cryptology Conference*, pages 445–462. Springer, 2006.

8 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

9 Amir Pnueli. On the extremely fair treatment of probabilistic algorithms. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 278–290, 1983.

10 Michael O Rabin. Probabilistic algorithms in finite fields. *SIAM Journal on computing*, 9(2):273–280, 1980.

11 Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

Brief Announcement: Sinkless Orientation Is Hard Also in the Supported LOCAL Model

Janne H. Korhonen
IST Austria, Klosterneuburg, Austria

Ami Paz
Faculty of Computer Science, Universität Wien, Austria

Joel Rybicki
IST Austria, Klosterneuburg, Austria

Stefan Schmid
Faculty of Computer Science, Universität Wien, Austria
Fraunhofer Institute for Secure Information Technology (SIT), Darmstadt, Germany

Jukka Suomela
Aalto University, Finland

Abstract

We show that any algorithm that solves the sinkless orientation problem in the supported LOCAL model requires $\Omega(\log n)$ rounds, and this is tight. The supported LOCAL is at least as strong as the usual LOCAL model, and as a corollary this also gives a new, short and elementary proof that shows that the round complexity of the sinkless orientation problem in the deterministic LOCAL model is $\Omega(\log n)$.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Supported LOCAL model, sinkless orientation, round elimination

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.58

Related Version *Full Version*: http://arxiv.org/abs/2108.02655

Funding *Janne H. Korhonen*: Project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 805223 ScaleML).
Ami Paz: We acknowledge the Austrian Science Fund (FWF) and netIDEE SCIENCE project P 33775-N.
Stefan Schmid: Research supported by the Austrian Science Fund (FWF) project ADVISE, I 4800-N, 2020-2023.

1 Introduction

Sinkless orientations. In the *sinkless orientation problem*, the task is to orient the edges of a graph such that all nodes of degree at least 3 have out-degree at least 1. The problem is always solvable, and easy to solve in the centralized setting, but a lot more challenging to solve efficiently in distributed or parallel settings.

The sinkless orientation problem is the canonical example of a problem that has round complexity $\Theta(\log \log n)$ rounds in the randomized LOCAL model but $\Theta(\log n)$ rounds in the deterministic LOCAL model. It is a rare example of a *locally checkable problem* in which randomness helps exponentially, and also an example of a locally checkable problem with an *intermediate* complexity – not solvable in $O(\log^* n)$ rounds but solvable in sub-diameter time.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 58; pp. 58:1–58:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Supported LOCAL model. While the complexity landscape in the usual LOCAL model is nowadays well understood [1, 3–5, 7, 8, 10, 11], and also some *weaker* models of distributed computing have been already explored [2, 9, 14], it has been wide open how the landscape changes when we switch to *stronger* models of computing. In this work we focus on the *supported* LOCAL model, which is strictly stronger than the LOCAL model.

In the supported LOCAL model [12, 15], the communication network $G = (V, E)$ and the unique identifiers are all known to all nodes, and the input is a subgraph H of G. That is, each node v receives as input *the entire structure* of the communication network G, including all the unique identifiers, and a list of its incident edges in H; we refer to the latter edges as *input edges.* Otherwise, the computation proceeds as in the standard LOCAL model, using all edges of G for communication. In our case, we would like to find a sinkless orientation in the input graph H.

The availability of the underlying globally-known communication graph G (a.k.a. the *support*) helps a lot with many problems. For example, all locally checkable problems with complexity $O(\log^* n)$ admit constant-time algorithms in the supported model – in essence, the support can be used to *break symmetry for free* [12]. Also if we had the promise that the support G is a *tree*, then the sinkless orientation problem would become trivial: we can orient all edges of G towards a leaf, and this orientation is also a valid orientation for any subgraph H. However, in this work we show that this trick only works in trees – we show that if, for example, G is a 5-regular graph, then the support is essentially useless.

The supported LOCAL model was introduced in the context of *software-defined networks* (SDNs). The underlying idea is that the communication graph G represents the unchanging physical network, and the input graph represents the logical state of the network to which the control plane (here, distributed algorithm) needs to respond to; see reference [15] for more details. However, supported LOCAL has proven useful as a theoretical model for lower bounds (this work, Foerster et al. [12], and the very recent work of Haeupler et al. [13]).

Our contributions. We prove that the round complexity of the sinkless orientation problem in the deterministic supported LOCAL model is $\Omega(\log n)$ rounds. By prior work, we also know that this is tight: the problem is solvable in $O(\log n)$ rounds (with or without support). Furthermore, the same problem can be solved in the randomized supported LOCAL model in $O(\log \log n)$ rounds.

In particular, we learn that in the supported LOCAL model there are locally checkable problems in which randomness helps exponentially. As a corollary, we cannot use the support to efficiently derandomize algorithms.

Relation to prior work. As a by-product, our work gives a new, short and elementary proof that shows that the round complexity of the sinkless orientation problem in the deterministic LOCAL model is $\Omega(\log n)$.

The standard proof is somewhat long and complicated. It builds on the *round elimination* technique [1, 7, 8], but round elimination has so far been unable to handle unique identifiers. Hence in prior work one has always taken a detour: first prove an $\Omega(\log \log n)$ lower bound in the randomized model without unique identifiers [8], and then apply the *deterministic gap result* of Chang et al. [10] to derive a deterministic $\Omega(\log n)$ lower bound.

By switching to the supported LOCAL model, we can give a direct proof without any detours through randomness and gap results. We directly show with elementary arguments that the complexity of sinkless orientation is $\Omega(\log n)$, both in the usual LOCAL model and also in the supported LOCAL model.

The underlying idea is, in essence, the same as the *ID graph technique* from the very recent work by Brandt et al. [9]. The ID graph in their work plays a role similar to the support in our work. However, Brandt et al. aimed at proving a lower bound for randomized local computation algorithms, while our aim is at proving a lower bound for deterministic supported LOCAL algorithms. The key technical difference is that ID graphs [9] need to have a large chromatic number, while our proof goes through even if the support is a bipartite graph. On the other hand, we need to do more work in the base case when we argue that 0-round algorithms do not exist.

2 Sinkless orientation lower bound

For technical convenience, we prove the result in a stronger *bipartite* version of the model. In bipartite supported LOCAL, we are given a promise that the support graph G is bipartite, and a 2-coloring is given to the nodes as an input; we refer to the two colors as black and white. We consider either the black or white nodes to be *active*, and the other color to be *passive*. All nodes of the graph run an algorithm as per supported LOCAL model; upon termination of the algorithm, the active nodes produce an output, and the passive nodes output nothing.

The outputs of the active nodes must form a globally valid solution; in particular, in sinkless orientation, the outputs of the active nodes already orients all edges, and both active and passive nodes of degree at least 3 must not be sinks. Note that any (supported) LOCAL algorithm can be turned into a bipartite algorithm with no round overhead, by running the algorithm as is and discarding the outputs of passive nodes.

We summarise the key lemmas of the proof next. For the full technical details and proofs, we refer the reader to the full version of the paper.

▶ **Lemma 1.** *Let G be a fixed 5-regular bipartite graph with girth g, and fixed unique identifiers and 2-coloring of the nodes. Let $0 < T < g/2$, and assume there is a T-round bipartite supported* LOCAL *algorithm $\mathcal{A}_T$ that solves sinkless orientation on G. Then there is a $(T-1)$-round bipartite supported* LOCAL *algorithm $\mathcal{A}_{T-1}$ that solves sinkless orientation on G.*

▶ **Lemma 2.** *Let G be a fixed 5-regular bipartite graph with girth g, and assume unique identifiers and 2-coloring on G are fixed. There is no algorithm solving sinkless orientation in bipartite supported* LOCAL *in 0 rounds on G.*

▶ **Theorem 3.** *Any deterministic algorithm solving sinkless orientation in the supported* LOCAL *model requires $\Omega(\log n)$ rounds.*

Proof. Let G be a bipartite 5-regular graph with girth $g = \Omega(\log n)$. Observe that we can obtain one e.g. by taking the bipartite double cover of any 5-regular graph of girth $\Omega(\log n)$, which are known to exist (see e.g., [6, Ch. 3]).

Assume that there is a supported LOCAL algorithm $\mathcal{A}_T$ that solves sinkless orientation in $T < g/2$ rounds on communication graph G. This implies that there is a bipartite supported LOCAL algorithm for sinkless orientation on G running in time T. By repeated application of Lemma 1, there is a sequence of bipartite supported LOCAL algorithms $\mathcal{A}_T, \mathcal{A}_{T-1}, \ldots, \mathcal{A}_1, \mathcal{A}_0$, where algorithm $\mathcal{A}_i$ solves sinkless orientation in i rounds. In particular, $\mathcal{A}_0$ solves sinkless orientation in 0 rounds. By Lemma 2, this is impossible, so algorithm $\mathcal{A}_T$ cannot exist. ◀

▶ **Corollary 4.** *Any deterministic algorithm solving sinkless orientation in the* LOCAL *model requires $\Omega(\log n)$ rounds.*

References

1 Alkida Balliu, Sebastian Brandt, Juho Hirvonen, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. Lower bounds for maximal matchings and maximal independent sets. In *Proc. 60th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2019)*, 2019.

2 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, Jan Studený, Jukka Suomela, and Aleksandr Tereshchenko. Locally checkable problems in rooted trees. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*, 2021.

3 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. Almost global problems in the LOCAL model. In *Proc. 32nd International Symposium on Distributed Computing (DISC 2018)*, 2018. doi:10.4230/LIPIcs.DISC.2018.9.

4 Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, 2020. doi:10.1145/3382734.3405715.

5 Alkida Balliu, Juho Hirvonen, Janne H Korhonen, Tuomo Lempiäinen, Dennis Olivetti, and Jukka Suomela. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC 2018)*, 2018. doi:10.1145/3188745.3188860.

6 Béla Bollobás. *Extremal graph theory.* Courier Corporation, 2004.

7 Sebastian Brandt. An Automatic Speedup Theorem for Distributed Problems. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, 2019. doi:10.1145/3293611.3331611.

8 Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symposium on Theory of Computing (STOC 2016)*, 2016. doi:10.1145/2897518.2897570.

9 Sebastian Brandt, Christoph Grunau, and Václav Rozhoň. The randomized local computation complexity of the Lovász local lemma. In *Proc. 40th ACM Symposium on Principles of Distributed Computing (PODC 2021)*, 2021.

10 Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. In *Proc. 57th IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, 2016. doi:10.1109/FOCS.2016.72.

11 Yi-Jun Chang and Seth Pettie. A Time Hierarchy Theorem for the LOCAL Model. *SIAM Journal on Computing*, 48(1):33–69, 2019. doi:10.1137/17M1157957.

12 Klaus-Tycho Foerster, Juho Hirvonen, Stefan Schmid, and Jukka Suomela. On the Power of Preprocessing in Decentralized Network Optimization. In *Proc. IEEE Conference on Computer Communications (INFOCOM 2019)*, 2019. doi:10.1109/INFOCOM.2019.8737382.

13 Bernhard Haeupler, David Wajc, and Goran Zuzic. Universally-optimal distributed algorithms for known topologies. In *STOC*, pages 1166–1179. ACM, 2021.

14 Will Rosenbaum and Jukka Suomela. Seeing Far vs. Seeing Wide: Volume Complexity of Local Graph Problems. In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC 2020)*, 2020. doi:10.1145/3382734.3405721.

15 Stefan Schmid and Jukka Suomela. Exploiting locality in distributed SDN control. In *Proc. ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013)*, pages 121–126. ACM Press, 2013. doi:10.1145/2491185.2491198.

Brief Announcement: Simple Majority Consensus in Networks with Unreliable Communication

Ariel Livshits
Technion – Israel Institute of Technology, Haifa, Israel

Yonatan Shadmi
Technion – Israel Institute of Technology, Haifa, Israel

Ran Tamir (Averbuch)
Technion – Israel Institute of Technology, Haifa, Israel

Abstract

In this work, we consider a synchronous model of n faultless agents, with a complete communication graph and messages that are lost with some constant probability $q \in (0, 1)$. In this model we show that there exists a protocol, called the Simple Majority Protocol, that solves consensus in 3 communication rounds with probability of agreement converging to 1 as $n \to \infty$. We also prove that 3 communication rounds are necessary for the SMP to achieve consensus, with high probability.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Majority consensus, probabilistic message loss, distributed systems

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.59

Related Version Simple Majority Consensus in Networks with Unreliable Communication
Full Version: `https://arxiv.org/pdf/2104.04996.pdf` [6]

Acknowledgements The authors thank Yoram Moses for his invaluable comments and suggestions.

1 Introduction

In the binary consensus problem, every agent is initially assigned some binary value, referred to as the agent's initial value. The goal of a protocol that solves consensus is to have every agent eventually decide on the same value, thus reaching agreement throughout the system. More formally, given any initial configuration of the agents, every run of a protocol that solves consensus must exhibit these three properties: *Decision* – every agent eventually decides on some value $v \in \{0, 1\}$, *Agreement* – if some agent has decided on v, no value other than v can be decided on by any other agent, and *Validity* – if some agent has decided on v, then v was initially assigned to some agent.

In the pursuit of developing distributed protocols for consensus, much of the literature routinely makes two powerful assumptions. The first is that communication links are reliable, and the second is that there exists an upper bound on the transmission delay of messages from one agent to another [1]. Nonetheless, communication networks are notoriously unreliable in many practical systems [2]. In fact, actual communication links may suffer from sudden crashes, resulting in messages in transit being lost forever.

In practice, the reliability and upper bound assumptions do not hold simultaneously. However, it has been proven that consensus is not solvable without assuming reliable communication [5], and without a bound on message delivery times, protocols solving consensus cannot provide any bound on when they halt. Alternatively, we may consider a model where there exists a stationary probability distribution on the event that messages are delivered by a certain deadline, but in such a model we may only provide probabilistic

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 59; pp. 59:1–59:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

guarantees for consensus. Fortunately, the growth in scale of modern distributed systems (e.g. cryptocurrency) has provided ample incentive for work in probabilistic models, given that large numbers naturally lend themselves well to probabilistic analysis.

We consider a model of n agents connected via a complete communication graph. Namely, we assume that each agent has an active communication channel to every other agent in the system. This is a valid approximation for unstructured overlays in peer to peer networks, e.g., Freenet, Gnutella and Fast Track [4]. Also, we assume that each agent has access to a synchronized global clock. This permits time to be split into equal-length communication rounds, in which all messages between agents are sent at the beginning of a communication round, and either arrive by the end of the round or are considered lost. Additionally, we assume that the agents themselves are faultless, and that all message loss events are i.i.d. with some constant probability $q \in (0, 1)$.

The Simple Majority Protocol (SMP) is a round-based, majority-rule protocol that attempts to solve majority consensus, a stronger variation of consensus, in which the validity clause stipulates that if a majority of agents were initially assigned the same value, then all agents must decide on this value. The SMP can be described as follows: In each round, every agent sends its current value to all other agents. Then, it waits to receive messages from all other agents. If a majority of received messages propose the same value, then the agent adopts this value for the next round. All ties are reconciled by readopting the agent's current value. After a fixed number of rounds $\boldsymbol{r}$, each agent decides on its currently adopted value.

In this work, we prove that the SMP solves consensus in 3 communication rounds with probability of agreement converging to 1 as $n \to \infty$ (i.e., with high probability, denoted by *w.h.p.*). We also prove that if the relative majority (defined in this work as the excess over 50%) of the initial values of agents is of the order of at least $\sqrt{n}$, then the SMP solves majority consensus *w.h.p.* in at most 2 communication rounds. Additionally, we show that 3 communication rounds are not only sufficient, but also necessary for the SMP to achieve consensus, with high probability.

2 Main Results

We begin by defining a set of useful notations. Let S_n be a system of n agents, and let $c(S_n)$ be the set of all 2^n possible configurations of S_n. Let $\sigma = (c_1, c_2, \ldots, c_n, \ldots)$ be an infinite sequence of configurations, where the k-th element of the sequence is some configuration of the system S_k, i.e., $\forall k \geq 1 : c_k \in c(S_k)$. Let $I_0(c_n)$ and $I_1(c_n)$ be the number of zeros and ones, respectively, in the configuration c_n. The relative majority of a configuration is then defined as $\delta(c_n) = \max\{I_0(c_n), I_1(c_n)\} - \left\lfloor \frac{n}{2} \right\rfloor$, i.e., the excess over 50% of the majority value. We will also refer to the sequence of relative majorities $\delta(\sigma) = (\delta(c_1), \delta(c_2), \ldots, \delta(c_n), \ldots)$, in which the k-th element is the relative majority of the k-th configuration in σ. We say that $\delta(\sigma) \in \Omega(\sqrt{n})$ *w.h.p.* holds for some sequence $\sigma = (c_1, c_2, \ldots, c_n, \ldots)$, if for any $\epsilon > 0$ there exists some constant $a > 0$ and index $k \in \mathbb{N}$ such that $\forall n \geq k : \mathbb{P}\left(\delta(c_n) \geq a \cdot \sqrt{n}\right) \geq 1 - \epsilon$. Similarly, $\delta(\sigma) \in \omega(\sqrt{n})$ *w.h.p.* holds if $\forall n \geq k : \mathbb{P}\left(\delta(c_n) > a \cdot \sqrt{n}\right) \geq 1 - \epsilon$. We say that $\delta(\sigma) \in \Omega(\sqrt{n})$ (or $\delta(\sigma) \in \omega(\sqrt{n})$) if this holds also for $\epsilon = 0$.

The SMP defines *a priori* the number of rounds r until termination. Denote the event that the SMP reaches consensus from an initial configuration c_n in r rounds by $\mathcal{C}\left(c_n, r\right)$. Similarly, we denote by $\mathcal{C}^m\left(c_n, r\right)$ the event that the SMP reaches majority consensus. The probability spaces over which our results hold are $\left\{(R(c_n), 2^{R(c_n)}, \mathbb{P})\right\}_{n \geq 1}$ where $R(c_n)$ is the set of all runs of a system of n agents, that start in the initial configuration c_n, and $\mathbb{P}(\cdot)$ is the natural probability measure on runs induced by the distribution on message loss, i.e. Bernoulli with parameter q (as done in, e.g., [3]).

▶ **Lemma 1.** *For any sequence of initial configurations $\sigma = (c_1, c_2, \ldots, c_n, \ldots)$, let $\tilde{\sigma} = (\tilde{c}_1, \tilde{c}_2, \ldots, \tilde{c}_n, \ldots)$ be the next sequence of configurations reached after a single round of the SMP. Then, $\delta(\tilde{\sigma}) \in \Omega(\sqrt{n})$ w.h.p. holds for $\tilde{\sigma}$.*

The lemma states that as the scale of the system approaches infinity, the relative majority of the configuration reached after a single round of the SMP is of the order of at least $\sqrt{n}$, regardless of the initial configuration of the system. To gain an intuitive understanding of this result, observe the worst case scenario. Specifically, consider the case where the initial configuration of the system is perfectly symmetric, i.e., the number of ones equals the number of zeros. In this case, after a single communication round, every agent receives approximately the same number of messages reporting zeros and ones. Thus, we assume that the configuration of the agents at the end of the first round is approximately an i.i.d. random vector whose components are Bernoulli random variables with parameter 0.5. The central limit theorem states that $\frac{1}{\sqrt{n}} \cdot \delta(\tilde{c}_n)$ converges in distribution to a Gaussian random variable. This means that $\frac{1}{\sqrt{n}} \cdot \delta(\tilde{c}_n) \in \Theta(1)$ *w.h.p.*, and hence $\delta(\tilde{c}_n) \in \Theta(\sqrt{n})$ *w.h.p.*, which implies that $\delta(\tilde{\sigma}) \in \Omega(\sqrt{n})$ *w.h.p.* (see Proposition 3 in [6] for a formal proof).

▶ **Lemma 2.** *Let $\sigma = (c_1, c_2, \ldots, c_n, \ldots)$ be a sequence of initial configurations. Then:*

- *If $\delta(\sigma) \in \Omega(\sqrt{n})$, then $\mathbb{P}\left[\mathcal{C}^m\left(c_n, 2\right)\right] \xrightarrow{n\to\infty} 1$*
- *If $\delta(\sigma) \in \omega(\sqrt{n})$, then $\mathbb{P}\left[\mathcal{C}^m\left(c_n, 1\right)\right] \xrightarrow{n\to\infty} 1$*

The lemma implies that for initial configurations that, to begin with, have a significant relative majority to one of the values, then *w.h.p.* the SMP reaches majority consensus in one or two communication rounds, depending on how much larger the initial relative majority is than $\sqrt{n}$. However, majority consensus cannot be ensured *w.h.p.* for all possible initial configurations. Intuitively, if the initial relative majority is too weak (e.g., on the order of $log(n)$), then it is most likely that the random losses in the network will completely hide it. In fact, the state at the end of round 1 is probabilistically equivalent to a sequence of n fair coin tosses, and hence, the majority at the end of round 1 will switch sides with a probability of about one half (see Propositions 1 & 2 in [6] for a formal proof). Our main result is:

▶ **Theorem 3.** *Let $\sigma = (c_1, c_2, \ldots, c_n, \ldots)$ be a sequence of initial configurations matching to a sequence of systems with n agents that grow in size as $n \to \infty$. Then: $\mathbb{P}\left[\mathcal{C}\left(c_n, 3\right)\right] \xrightarrow{n\to\infty} 1$.*

This theorem asserts that for a sequence of systems of n agents that progressively grow larger with n, the SMP will reach agreement *w.h.p.*, after only 3 communication rounds, regardless of the initial configurations of those systems. This result seems intuitive for initial configurations that already possess a large relative majority towards some value, considering that it is very likely that the SMP will cause a small minority of agents to adopt the value of the majority. However, it is surprising that this result holds for initial configurations whose relative majority is close to 0, especially if the initial configuration is perfectly symmetric. However, Lemma 1 maintains that even in the worst case, the symmetry will break equiprobably to one of the values after a single round of the SMP, the relative majority will be of the order of at least $\sqrt{n}$, and then, by Lemma 2, consensus will be reached in at most 2 additional rounds (see Theorem 1 in [6] for a formal proof).

▶ **Theorem 4.** *Let $\sigma = (c_1, c_2, \ldots, c_n, \ldots)$ be a sequence of initial configurations such that for all $c_n \in \sigma$ it holds that $I_0(c_n) = I_1(c_n)$. Then: $\mathbb{P}\left[\mathcal{C}\left(c_n, 2\right)\right] \xrightarrow{n\to\infty} 0$.*

The theorem provides a lower bound of 2 rounds for the SMP to reach consensus. Specifically, if the initial configuration of the system is perfectly symmetric, then 3 rounds are not only sufficient, but also necessary (see Theorem 2 in [6] for a formal proof).

References

1 Marcos K Aguilera. Stumbling over consensus research: Misunderstandings and issues. In *Replication*, pages 59–72. Springer, 2010.

2 Rachid Guerraoui, Michel Hurfinn, Achour Mostéfaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in asynchronous distributed systems: A concise guided tour. In *Advances in Distributed Systems*, pages 33–47. Springer, 2000.

3 Joseph Y Halpern and Mark R Tuttle. Knowledge, probability, and adversaries. *Journal of the ACM (JACM)*, 40(4):917–960, 1993.

4 Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

5 Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

6 Ran Tamir, Ariel Livshits, and Yonatan Shadmi. Simple majority consensus in networks with unreliable communication. *arXiv preprint*, 2021. `arXiv:2104.04996`.

Brief Announcement: Crystalline: Fast and Memory Efficient Wait-Free Reclamation

Ruslan Nikolaev
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Binoy Ravindran
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

Abstract

We present a new wait-free memory reclamation scheme, Crystalline, that simultaneously addresses the challenges of high performance, high memory efficiency, and wait-freedom. Crystalline guarantees complete wait-freedom even when threads are dynamically recycled, asynchronously reclaims memory in the sense that any thread can reclaim memory retired by any other thread, and ensures (an almost) balanced reclamation workload across all threads. The latter two properties result in Crystalline's high performance and high memory efficiency, a difficult trade-off for most existing schemes. Our evaluations show that Crystalline exhibits outstanding scalability and memory efficiency, and achieves superior throughput than state-of-the-art reclamation schemes as the number of threads grows.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases memory reclamation, wait-free, reference counting, hazard pointers

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.60

Related Version *Full Version*: `https://arxiv.org/abs/2108.02763`

Supplementary Material *Software (Source Code)*: `https://github.com/rusnikola/wfsmr`

Funding ONR grants: N00014-18-1-2022, N00014-19-1-2493, and AFOSR grant: FA9550-16-1-0371.

Acknowledgements We thank the anonymous reviewers for their invaluable feedback.

1 Introduction

Non-blocking data structures do not use simple mutual exclusion: a concurrent thread may hold an obsolete pointer to an object which is about to be freed by another thread. Responding to this challenge, *safe memory reclamation* (SMR) schemes for unmanaged C/C++ code have been proposed in the literature (e.g., [1, 2, 3, 4, 5, 6, 9, 11, 12, 13, 14, 16]). However, they typically involve major trade-offs, e.g., memory efficiency vs. throughput.

The significance of balancing the reclamation workload – the task of reclaiming deleted memory objects – across all threads have not received adequate attention in the literature. Consider the common scenario when read operations dominate, but data is still modified. If the reclamation workload is unbalanced, as in most existing reclamation schemes [4, 6, 12, 16], then most threads are not actively reclaiming memory, which can cause memory waste.

Vast majority of SMR schemes (e.g., [6, 12, 16]) are inherently *synchronous*, i.e., they need to periodically examine which objects marked for deletion can be safely freed. In contrast, reference counting [7, 15] is *asynchronous*: an *arbitrary* thread with the last reference frees an object. Unfortunately, reference counting is often impractical due to very high overheads when accessing objects. Hyaline [8, 11], an approach where reference counters are only used when objects are retired, is still *asynchronous* and exhibits high performance. However, Hyaline can be blocking since its memory usage is unbounded when threads starve.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 60; pp. 60:1–60:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

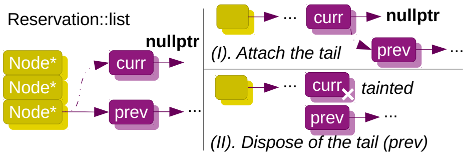


Figure 1 Crystalline-W: list tainting.

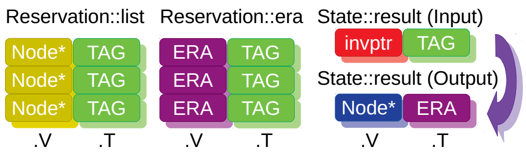


Figure 2 Crystalline-W: reservations and state.

We propose Crystalline-W, which has the following properties: wait-freedom, asynchronous reclamation, and balanced reclamation workload. Crystalline-W extends a lock-free algorithm, Crystalline-L, which we also present in the paper (Section 2). Crystalline-L, in turn, is based on Hyaline-1S [8, 11], a prior scheme. Making Crystalline wait-free involves overcoming a set of unique challenges, caused in part due to asynchronous memory reclamation (Section 3).

2 Lock-Free Reclamation with Crystalline-L

In Hyaline [8, 11], threads explicitly annotate each operation. When objects are detached from a data structure, they are first *retired* and then freed when it is safe to do so. Hyaline-1S is a variant that bounds memory usage for *stalled* threads by explicitly tracking local pointers via a special `protect` method using the global era clock [12, 16]. However, Hyaline-1S is still not lock-free unless operations are periodically restarted for *starving* threads. The problem arises when a thread reserves an increasing number of local pointers in an unbounded loop (e.g., one "unlucky" thread is stuck traversing a list because it keeps growing).

The culprit is a simplistic API [16] which enables retrieving an unbounded number of local pointers. To avoid this issue, alternative APIs [6, 12] explicitly differentiate each local pointer reservation in `protect`. Crystalline-L modifies Hyaline-1S so that each thread has several (rather than just one) reservations. Each reservation has its own list of retired objects.

Hyaline-1S retires an entire *batch* of objects, and each batch is attached to every active reservation. Each object reserves space for a list of retired objects. Whereas Hyaline-1S only needs MAX_THREADS+1 objects to successfully retire a batch, Crystalline-L handles MAX_IDX local pointers and consequently needs MAX_THREADS×MAX_IDX+1 objects.

One problem with Hyaline-1S is that a batch must aggregate *all* objects before retirement can even start, which is further aggravated in Crystalline-L. In practice, the required number of objects is much lower as each object is appended to the respective list only if the list's era overlaps with the batch's minimum birth era. Crystalline-L uses *dynamic* batches to avoid their excessive growth. The `retire` method first checks how many lists are to be changed for the batch to be fully retired and records the location of the corresponding reservations. If the number of objects in the batch suffices, `retire` completes by appending the objects to their corresponding lists. Otherwise, `retire` is repeated later when more objects are available.

3 Wait-Free Reclamation with Crystalline-W

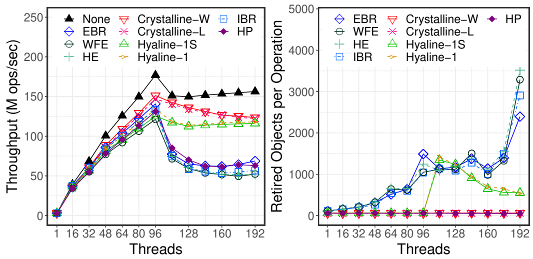
The Crystalline-L scheme is still only lock-free because of two fundamental challenges:

- `retire` has an unbounded loop: `protect` or another `retire` contends on the same list.
- `protect` has an unbounded loop which must converge on the era value; however, the era clock unconditionally increments when a new object is allocated.

To solve the first problem, we use unconditional SWAP in `retire` as shown in Figure 1. We initialize the *next* field of a retired object with `nullptr`, and swap the current list head with the pointer to the retired object. Putting corner cases aside, two major cases exist. If

None	no reclamation (leak memory)
Hyaline-1	basic variant [8, 11]
Hyaline-1S	supports stalled threads [8, 11]
HP	the hazard pointers scheme [6]
HE	the hazard eras scheme [12]
IBR	2GEIBR (interval-based) [16]
WFE	the wait-free eras scheme [9]
EBR	epoch-based reclamation

Figure 3 Evaluated reclamation schemes.



(a) Throughput. **(b)** Retired objs.

Figure 4 Lock-free hash map (read-dominated).

the retired object still has its next field intact, it simply attaches the previous list as its tail (part I). It is also possible that a thread associated with the reservation already started traversing and dereferencing retired objects. Crystalline-W additionally taints the retrieved *next* pointer of all traversed objects by using SWAP. `retire` finds that the next field is now tainted (part II). Thus, `retire` traverses the tail on behalf of the thread that tainted *next*.

The second problem was already considered by WFE [9] for hazard eras [12]. The approach, which we also adopt in Crystalline-W, is based on a fast-path-slow-path idea to coordinate global era clock increments. Each thread maintains its *state* for the slow path, which is taken when `protect` fails to retrieve a pointer after a small number of steps. The *result* field of *state* is used for both input and output. On input, a current slow path cycle is advertised. Output contains a retrieved pointer with the corresponding era. In Figure 2, we demonstrate how input and output values must be aligned. Reservations need to be extended with extra tags which identify slow path cycles and prevent spurious updates. Despite similarities, Crystalline-W substantially diverges from the original WFE's idea due to its unique retirement mechanism. Complete details can be found in [10].

4 Evaluation

Crystalline's implementation extends the benchmark of [9, 11, 16]. We evaluated all schemes (Figure 3) from 1 to 192 threads on a 96-core machine consisting of four Intel Xeon E7-8890 v4 2.20 GHz CPUs, 256GB of RAM. We present results for hash map [6] under (90% get, 10% put) workload. Complete details of our experiments and additional results are in [10].

Crystalline-L/-W achieve superior throughput (Figure 4a), which is especially evident under oversubscription, where the gap with other algorithms is as large as 2x. Hyaline-1/-1S's throughput is worse, which is due to a smaller granularity of reservations. WFE has the worst throughput. Crystalline-L/-W achieve exceptional memory efficiency which is on par with HP (Figure 4b). Even Hyaline-1/1S are visibly less memory efficient.

5 Conclusions

We presented a new wait-free SMR scheme, Crystalline-W, which guarantees complete wait-freedom with bounded memory usage. Crystalline-W is based on a lock-free algorithm, Crystalline-L, which is also presented in the paper. Crystalline-L is an improved version of Hyaline-1S that additionally guarantees bounded memory usage even when threads starve. Both Crystalline-L/-W show very high throughput and high memory efficiency, unparalleled among existing schemes, which is especially evident in read-dominated workloads.

References

1 Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. Concurrent Deferred Reference Counting with Constant-Time Overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, pages 526–541. ACM, 2021. doi:10.1145/3453483.3454060.

2 Nachshon Cohen. Every Data Structure Deserves Lock-free Memory Reclamation. *Proc. ACM Program. Lang.*, 2(OOPSLA):143:1–143:24, 2018. doi:10.1145/3276513.

3 Andreia Correia, Pedro Ramalhete, and Pascal Felber. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 205–218. ACM, 2021. doi:10.1145/3437801.3441596.

4 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

5 Jeehoon Kang and Jaehwang Jung. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, pages 314–328. ACM, 2020. doi:10.1145/3385412.3385978.

6 Maged M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. doi:10.1109/TPDS.2004.8.

7 Maged M. Michael and Michael L. Scott. Correction of a Memory Management Method for Lock-Free Data Structures. Technical report, University of Rochester, USA, 1995. URL: https://www.cs.rochester.edu/u/scott/papers/1995_TR599.pdf.

8 Ruslan Nikolaev and Binoy Ravindran. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 419–421. ACM, 2019. doi:10.1145/3293611.3331575.

9 Ruslan Nikolaev and Binoy Ravindran. Universal Wait-Free Memory Reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 130–143. ACM, 2020. doi:10.1145/3332466.3374540.

10 Ruslan Nikolaev and Binoy Ravindran. Crystalline: Fast and Memory Efficient Wait-Free Reclamation (full paper, arXiv), 2021. arXiv:2108.02763.

11 Ruslan Nikolaev and Binoy Ravindran. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, pages 987–1002. ACM, 2021. doi:10.1145/3453483.3454090.

12 Pedro Ramalhete and Andreia Correia. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, pages 367–369. ACM, 2017. doi:10.1145/3087556.3087588.

13 Gali Sheffi, Maurice Herlihy, and Erez Petrank. VBR: Version Based Reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, pages 443–445. ACM, 2021. doi:10.1145/3409964.3461817.

14 Ajay Singh, Trevor Brown, and Ali Mashtizadeh. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 175–190. ACM, 2021. doi:10.1145/3437801.3441625.

15 John D. Valois. Lock-free Linked Lists Using Compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222. ACM, 1995. doi:10.1145/224964.224988.

16 Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 1–13. ACM, 2018. doi:10.1145/3178487.3178488.

Brief Announcement: Design and Verification of a Logless Dynamic Reconfiguration Protocol in MongoDB Replication

William Schultz ✉
Northeastern University, Boston, MA, USA

Siyuan Zhou ✉
MongoDB, New York, NY, USA

Stavros Tripakis ✉
Northeastern University, Boston, MA, USA

Abstract

We introduce a novel dynamic reconfiguration protocol for the MongoDB replication system that extends and generalizes the single server reconfiguration protocol of the Raft consensus algorithm. Our protocol decouples the processing of configuration changes from the main database operation log, which allows reconfigurations to proceed in cases when the main log is prevented from processing new operations. Additionally, this decoupling allows for configuration state to be managed by a *logless* replicated state machine, storing only the latest version of the configuration and avoiding the complexities of a log-based protocol. We present a formal specification of the protocol in TLA+, initial verification results of model checking its safety properties, and an experimental evaluation of how reconfigurations are able to quickly restore a system to healthy operation when node failures have stalled the main operation log. This announcement is a short version and the full paper is available at [16].

2012 ACM Subject Classification Information systems → Parallel and distributed DBMSs; Software and its engineering → Software verification

Keywords and phrases Reconfiguration, Consensus, State Machine Replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.61

Related Version *Full Version*: `https://arxiv.org/abs/2102.11960`

1 Introduction

Distributed replication systems based on the replicated state machine model [13] have become ubiquitous as the foundation of modern, fault-tolerant data storage systems. In order for these systems to ensure availability in the presence of faults, they must be able to dynamically replace failed nodes with healthy ones, a process known as dynamic reconfiguration. The protocols for building distributed replication systems have been well studied and implemented in a variety of systems [2, 20, 4, 18]. Paxos [5] and, more recently, Raft [11], have served as the logical basis for building provably correct distributed replication systems. Dynamic reconfiguration, however, is an additionally challenging and subtle problem [1] that has not been explored as extensively as the foundational consensus protocols underlying these systems. The Raft protocol, originally published in 2014, provided a dynamic reconfiguration algorithm in its initial publication, but did not include a precise discussion of its correctness or include a formal specification or proof. A critical safety bug [10] in one of its reconfiguration protocols was found after initial publication, which has since been fixed. The discovery of bugs like these, however, demonstrates that the design and verification of a safe dynamic reconfiguration protocol is a non-trivial task.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 61; pp. 61:1–61:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Since its inception, MongoDB [9], a general purpose distributed database, included a replication system [15] with a mechanism for clients to dynamically reconfigure replica membership. This legacy reconfiguration protocol was, however, unsafe in certain cases. In recent versions of MongoDB, reconfiguration has become a more common operation, necessitating the need for a redesigned, safe reconfiguration protocol with provable correctness guarantees. It was also desirable that this new protocol minimize changes to the existing, legacy protocol, which did not use a log-based approach to managing configurations. In this brief announcement we propose *MongoRaftReconfig*, a redesigned, safe reconfiguration protocol with provable correctness guarantees that minimizes changes to the existing, legacy protocol where possible. *MongoRaftReconfig* provides *logless* dynamic reconfiguration by decoupling the processing of configuration changes from the main database operation log, and improves upon and generalizes the single server reconfiguration protocol of standard Raft. This announcement is a short version of the full paper available at [16].

2 *MongoRaftReconfig*: A Logless Dynamic Reconfiguration Protocol

Many consensus based replication protocols [17, 7, 11] utilize the main operation log (referred to as the *oplog*, in MongoDB) to manage configuration changes by writing special reconfiguration log entries. *MongoRaftReconfig* instead decouples configuration updates from the main operation log by managing the configuration state of a replica set in a separate replicated state machine, which we refer to as the *config log*. The config log is maintained alongside the oplog, and manages the configuration state used by the overall protocol.

Decoupling these two conceptually distinct logs, the oplog and the config log, enables certain optimizations and simplifications in *MongoRaftReconfig* that would not be possible in a protocol where both logs are interleaved with each other. First, it allows for a simplification of the config log structure by observing that configuration changes are an "update only" operation. This obviates the need to store the entire log history, allowing the config log to operate as a *logless* replicated state machine, storing only the latest version of the configuration state. Second, it prevents the dynamics of either log negatively impacting the other unnecessarily. For example, it is possible to commit database writes in either log independently, without requiring previous writes in the other log to also become committed. This can allow the config log to bypass the oplog, allowing for reconfigurations in cases where a slow or stalled oplog replication channel would otherwise prevent reconfigurations from proceeding. For a detailed description of the protocol and its behaviors see [16].

3 Formal Specification and Model Checking

To formally describe and model check safety properties of *MongoRaftReconfig*, we use the TLA+ language [6]. TLA+ is a formal specification language for describing distributed and concurrent systems that is based on first order and temporal logic [12]. The full TLA+ specification of *MongoRaftReconfig* can be found at [14]. Note that the TLA+ language does not impose an underlying system or communication model (e.g. message passing, shared memory, etc.), which allows one to write specifications at a wide range of abstraction levels. Our specifications are written at a deliberately high level of abstraction, ignoring some lower level details of the protocol and system model. In practice, we have found the abstraction level of our specifications most useful for understanding and communicating the essential behaviors and safety characteristics of the protocol, while also serving to make automated verification feasible.

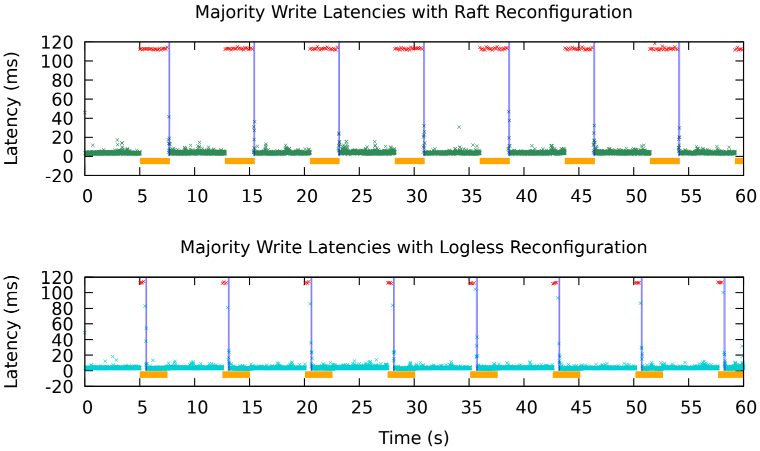


Figure 1 Latency of majority writes in the face of node degradation and reconfiguration to recover. Red points indicate writes that timed out. Orange bars indicate intervals of time where system entered a *degraded* mode. Vertical blue bars indicate completion of reconfiguration events.

For initial verification, we undertook an automated approach using TLC [19], an explicit state model checker for TLA+ specifications. We verified finite instances of the protocol to provide a sound guarantee of protocol correctness for fixed, finite parameters. It has been observed elsewhere [8] that relatively small, finite instances of distributed protocols are often sufficient to exhibit behaviors that are generalizable to larger (potentially infinite) instances, which helps to provide initial confidence in protocol correctness. This verification approach is, however, incomplete, in the sense that it does not establish correctness of the protocol for an unbounded number of servers. A goal for future work is to develop a general safety proof using the TLA+ proof system [3].

4 Experimental Evaluation

To demonstrate the benefits of *MongoRaftReconfig*, we designed an experiment to measure how quickly a replica set can reconfigure in new nodes to restore write availability when it faces periodic phases of degradation. For comparison, we implemented a simulated version of the Raft reconfiguration algorithm in MongoDB by having reconfigurations write a no-op oplog entry and requiring it to become committed before the reconfiguration can complete. Our experiment initiates a 5 node replica set and we periodically simulate a failure of two nodes, reconfigure them out of the set, and add in two new, healthy nodes. The ability of *MongoRaftReconfig* to quickly reconfigure in this scenario is seen in the results of Figure 1. Full details on the experimental setup are presented in [16].

References

1 Marcos Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 2010.

2 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1281100.1281103.

3 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010.

4 Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 2020. doi:10.14778/3415478.3415535.

5 Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 1998. doi:10.1145/279227.279229.

6 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, June 2002.

7 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

8 Haojun Ma, Aman Goel, Jean Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. I4: Incremental inference of inductive invariants for verification of distributed protocols. In *SOSP 2019 - Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019. doi:10.1145/3341301.3359651.

9 MongoDB Github Project, 2021. URL: https://github.com/mongodb/mongo.

10 Diego Ongaro. Bug in single-server membership changes, July 2015. URL: https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J.

11 Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, USA, 2014. USENIX Association.

12 Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

13 Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990. doi:10.1145/98163.98167.

14 William Schultz. MongoRaftReconfig TLA+ Specification, 2021. URL: https://github.com/will62794/logless-reconfig/blob/3d6b378eae7dabe3a35a0b4042bfc55fd178cb21/specs/MongoRaftReconfig.tla.

15 William Schultz, Tess Avitabile, and Alyson Cabral. Tunable consistency in mongodb. *Proceedings of the VLDB Endowment*, 12(12):2071–2081, 2019.

16 William Schultz, Siyuan Zhou, and Stavros Tripakis. Design and verification of a logless dynamic reconfiguration protocol in mongodb replication. *arXiv preprint*, 2021. arXiv:2102.11960.

17 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012*, 2019.

18 Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3386134.

19 Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

20 Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. PaxosStore: High-availability storage made practical in WeChat. In *Proceedings of the VLDB Endowment*, 2017. doi:10.14778/3137765.3137778.

Brief Announcement: Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption

Sravya Yandamuri
Duke University, Durham, NC, USA

Ittai Abraham
VMware Research, Herzliya, Israel

Kartik Nayak
Duke University, Durham, NC, USA

Michael Reiter
Duke University, Durham, NC, USA

Abstract

Small trusted hardware primitives can improve fault tolerance of Byzantine Fault Tolerant (BFT) protocols to one-half faults. However, existing works achieve this at the cost of increased communication complexity. In this work, we explore the design of communication-efficient BFT protocols that can boost fault tolerance to one-half without worsening communication complexity. Our results include a version of HotStuff that retains linear communication complexity in each view and a version of the VABA protocol with quadratic communication, both leveraging trusted hardware to tolerate a minority of corruptions. As a building block, we present communication-efficient provable broadcast, a core broadcast primitive with increased fault tolerance. Our results use expander graphs to achieve efficient communication in a manner that may be of independent interest.

2012 ACM Subject Classification Theory of computation → Communication complexity

Keywords and phrases communication complexity, consensus, trusted hardware

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.62

Related Version *Full Version*: `https://eprint.iacr.org/2021/184.pdf`

1 Introduction

The number of faults tolerated by a Byzantine Fault Tolerant (BFT) protocol depends on the network assumptions between parties, the use of cryptography, and other assumptions. In particular, it is known that to maintain safety when the system is asynchronous, without additional assumptions, one cannot tolerate one-third or more Byzantine faults [8]. However, tolerating fewer than one-third Byzantine faults may not be enough for some applications. There are two known approaches to increase this fault threshold. The first approach is to give up safety in asynchrony by assuming synchrony and using some method to limit the ability of the adversary to simulate honest parties (e.g. assuming a PKI or proof-of-work) [2,9,11]. The second approach lets the adversary delay messages but limits its ability to corrupt by assuming the existence of a *trusted hardware*. The adversary cannot tamper with this hardware even if it fully controls the party. At a high-level, the hardware provides non-equivocation guarantees, essentially transforming Byzantine failures to omission failures and hence improving the fault tolerance threshold to one-half (e.g., [6,8,10]) in partial synchrony and asynchrony.

In this work, we focus on the use of *small trusted hardware* primitives to tolerate a minority Byzantine corruption and stay safe in asynchrony. Specifically, each node is equipped with hardware that implements the abstraction of an "append-only log," the contents to which it can attest using a *conventional* digital signature with a key that it holds. This capability

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 62; pp. 62:1–62:4
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is supported by numerous *existing* trusted add-ons (TPMs [1], YubiKeys [14], smartcards, etc.) and is far simpler to implement than secure enclaves for arbitrary computation, as Intel SGX [7] attempts to do – but arguably fails [5, 13, 15, 16, 17].

The use of such small trusted hardware to boost fault tolerance was explored in A2M [6] and TrInc [12], which specifically improved PBFT [4]. However, this came at the expense of an $O(n^3)$ communication complexity per view for consensus among n parties, measured as the (expected) number of words that all honest parties send. On the other hand, in the standard setting, we have recently seen considerable progress in improving communication complexity of consensus protocols. In particular, HotStuff [19] achieves $O(n)$ communication complexity per view under partial synchrony and VABA [3] achieves the optimal $O(n^2)$ communication complexity under asynchrony. A natural question is whether fault tolerance can be boosted (but communication costs retained) in these protocols using small trusted hardware. In this abstract, we answer these questions affirmatively for a corruption threshold $t \leq (\frac{1}{2} - \epsilon)n$ and $\epsilon > 0$.

2 Communication-Efficient BFT with Small Trusted Hardware

We first describe a communication-efficient provable broadcast primitive that will be used as a building block towards communication-efficient variants of HotStuff and VABA tolerating $t \leq (\frac{1}{2} - \epsilon)n$ Byzantine faults. We only describe the intuition here; for a detailed explanation, refer to the full version of the paper [18].

Communication-Efficient Provable Broadcast. In provable broadcast, a designated leader sends a value to all the parties and obtains a proof that a majority of the parties delivered this value. For safety, the leader should only be able to obtain a proof for one value, and for liveness, an honest leader should obtain this value even without participation from faulty parties. Our goal is to achieve this primitive with linear communication complexity while having an $O(1)$-sized proof.

A straightforward approach would be to use the non-equivocation property of the hardware, i.e., the hardware can only produce a *signed attestation* for one value at one position in the log. Thus, intuitively, if $\frac{n}{2} + 1$ parties attest to a value at a position, then no other value can have $\frac{n}{2} + 1$ attestations. However, while a party's attestation from its trusted hardware is sufficient for safety, receiving such proofs from $O(n)$ parties produces an $O(n)$-sized proof sent to the leader. This does not satisfy the $O(1)$-sized proof requirement. Our solution relies on parties diffusing the attestations to a constant number of other parties, called their *neighbors*, instead of sending the attestations directly to the leader. A party sends a *vote* to the leader if it receives attestations from a threshold of its neighbors. This vote can be a threshold signature share, which can eventually be combined by the leader into an $O(1)$-sized voting proof. Why does this work? We connect parties to each other using a constant-degree expander graph [18]. Informally, to send a (non-attested) vote, a party just needs to verify that a constant fraction of its neighbors have attested. The expander graph construction guarantees that even if an ϵn fraction of the parties vote, a majority of the parties must have attested (ensuring safety). Similarly, to guarantee liveness, the graph can be parameterized to ensure sufficiently many parties vote if all honest parties attest

Results. Our first result improves HotStuff to tolerate a $t \leq (\frac{1}{2} - \epsilon)n$ corruption while still retaining its linear communication complexity per view.

▶ **Theorem 1** (HotStuff-M, Informal). *For any $\epsilon > 0$, there exists a primary-backup based BFT consensus protocol with $O(n)$ communication complexity per view consisting of n parties, each having a small trusted hardware, such that $t \leq (\frac{1}{2} - \epsilon)n$ of the parties are Byzantine.*

HotStuff is a primary-backup protocol that progresses in a sequence of views, each having a designated leader (primary) and consisting of three rounds. HotStuff routes all messages (votes) through the leader in each of these rounds while keeping the message size $O(1)$ for a total of $O(n)$ communication per view. Abstractly, this can be viewed as a sequence of provable broadcasts (though with additional $O(1)$-sized messages). HotStuff crucially relies on threshold signatures to aggregate votes of individual parties into an $O(1)$-sized message; these signatures act as a proof for parties in subsequent rounds/views to determine whether they should vote in that round.

To increase the fault tolerance of HotStuff, we replace the steady state of its protocol with three sequential provable broadcasts led by the leader of the view. However, this alone does not suffice for safety across views. In particular, while our trusted hardware provides us with an abstraction of an append-only log that disallows appending different values at the same position (equivocation), a party can potentially present only selected (older) entries of the log during a view change. This can potentially result in a safety or liveness violation. Of course, this could be fixed by always presenting the entire log each time, but the communication complexity would grow (unbounded) with the number of views. Instead, we use a combination of techniques including: multiple logs (though only $O(1)$), one for each phase of the protocol; tying log positions to view numbers; and using one attestation to present the end state of all logs.

Our second result improves the VABA protocol of Abraham et al. [3] to tolerate minority corruption while retaining its $O(n^2)$ communication complexity. We show the following:

▶ **Theorem 2** (VABA-M, informal). *For any $\epsilon > 0$, there exists a validated asynchronous Byzantine Agreement protocol with $O(n^2)$ communication complexity consisting of n parties, each having a small trusted hardware, such that $t \leq (1/2 - \epsilon)n$ parties are Byzantine.*

In each view of VABA, in parallel, each party attempts to drive progress by acting as a leader in a "proposal promotion" (similar to a view of HotStuff). After $n - t$ proposal promotions complete, the parties elect one leader randomly and adopt the progress from the leader's proposal promotion instance during the view-change step. Depending on whether the leader completed its proposal promotion, parties decide at the end of the view or try again in another view.

There are two key challenges in augmenting VABA to tolerate a minority corruption. The challenges relate to the amount of storage in the small trusted hardware and maintaining the communication complexity of the VABA protocol. First, in HotStuff-M, only $O(1)$ logs were used. A straightforward translation would require $O(n)$ logs. If reduced to $O(1)$ logs, each party needs to send $O(n)$ attestations to other parties during a single round of the protocol. The challenge relates to the existence of arbitrary message ordering across proposal promotion instances and the fact that the trusted hardware only supports an append-only log. Our solution crucially relies on the fact that the parties have the same neighbors across proposal promotion instances, and thus, even if values from proposal promotion instances are appended arbitrarily, parties can perform the necessary validation across instances. Second, the view-change step requires every party to share "progress" from the elected leader's proposal promotion instance to all parties. However, due to the concern described earlier, only a party's neighbors can validate whether it used the trusted hardware correctly. To make matters worse, a party or its neighbors can be Byzantine. Fortunately, since parties are connected using an expander graph, we can bound the number of parties with a majority of Byzantine neighbors. By careful analysis, we ensure the delivery of the latest state of the elected leader's proposal promotion instance to all parties. We explain formal details of these results in the full version of the paper [18].

References

1 Trusted computing group. URL: `https://trustedcomputinggroup.org/`.

2 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

3 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

4 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

5 Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy*, 2019.

6 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6):189–204, 2007.

7 Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

8 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. `doi:10.1145/42282.42283`.

9 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

10 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 295–308, 2012.

11 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.

12 Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.

13 Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on intel sgx. *arXiv preprint*, 2020. `arXiv:2006.13598`.

14 Suresh Thiru, Shamalee Deshpande, and Stina Ehrensvard. Yubikey strong two factor authentication, January 2021. URL: `https://www.yubico.com/`.

15 Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium.* USENIX Association, August 2018. See also technical report Foreshadow-NG [17].

16 Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM Conference on Computer and Communications Security*, 2017.

17 Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [15].

18 Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K Reiter. Communication-efficient bft protocols using small trusted hardware to tolerate minority corruption. *IACR Cryptol. ePrint Arch.*, 2021:184, 2021.

19 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint*, 2018. `arXiv:1803.05069`.

Brief Announcement: Ordered Reliable Broadcast and Fast Ordered Byzantine Consensus for Cryptocurrency

Pouriya Zarbafian
University of Sydney, Australia

Vincent Gramoli
University of Sydney, Australia
EPFL, Lausanne, Switzerland

Abstract

The problem of transaction reordering in blockchains, also known as the blockchain anomaly [11], can lead to fairness limitations [8] and front-running activities [6] in cryptocurrency. To cope with this problem despite $f < \frac{n}{3}$ byzantine processes, Zhang et al. [12] have introduced the *ordering linearizability* property ensuring that if two transactions or commands are perceived by all correct processes in the same order, then they are executed in this order. They proposed a generic distributed protocol that first orders commands and then runs a leader-based consensus protocol to agree on these orders, hence requiring at least 11 message delays. In this paper, we parallelize the ordering with the execution of the consensus to require only 6 message delays. For the ordering, we introduce the *ordered reliable broadcast* primitive suitable for broadcast-based cryptocurrencies (e.g., [3]). For the agreement, we build upon the DBFT leaderless consensus protocol [4] that was recently formally verified [1]. The combination is thus suitable to ensure ordering linearizability in consensus-based cryptocurrencies (e.g., [5]).

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases distributed algorithm, consensus, reliable broadcast, byzantine fault tolerance, linearizability, blockchain

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.63

Ordering Linearizability. Ordering linearizability [12] requires that command c_1 is ordered before another command c_2 if all the correct processes perceive c_1 before c_2. Zhang et al. have implemented ordering linearizability by exploiting the median value of the timestamps perceived by $2f+1$ distinct processes as an ordering indicator. We say that such a median value is *correctly bounded* as it is both upper bounded and lower bounded by the timestamps observed by correct processes.

Ordered Reliable Broadcast. To collect timestamps from $2f+1$ distinct processes, we modify the reliable broadcast protocol [2] in the asynchronous communication model to obtain a variant that preserves ordering linearizability. In our resulting *ordered reliable broadcast*, messages are delivered with an additional set of $2f+1$ signed timestamps. In order to not introduce any extra message delays, processes piggyback (i) a signed value of their clock in their ECHO messages, and (ii) a set of $2f+1$ signed timestamps in their READY messages (this set of $2f+1$ timestamps is collected from the ECHO messages received). As a result, messages that are delivered from the reliable broadcast come with a set T of $2f+1$ signed timestamps; the median timestamp of this set is correctly bounded and can be used as an ordering indicator that preserves ordering linearizability.

35th International Symposium on Distributed Computing (DISC 2021).
Editor: Seth Gilbert; Article No. 63; pp. 63:1–63:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

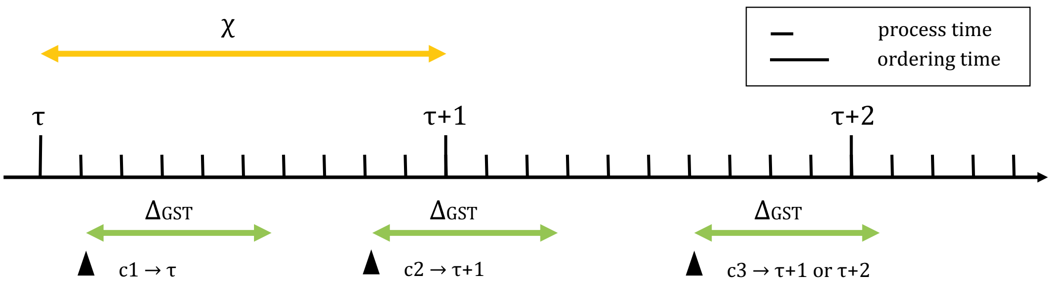


Figure 1 Command $c1$ will be sequenced with the ordering indicator τ, while command $c2$ will likely be sequenced with the ordering indicator $\tau + 1$. Because some correct processes may observe command $c3$ in $\tau + 1$ while others in $\tau + 2$, the ordering indicator computed by correct processes may vary for $c3$.

Agreeing on Ordering Indicators. Due to the combined effects of asynchrony and byzantine processes, the median value delivered by the ordered reliable broadcast is not necessarily equal at each process. To ensure agreement on the ordering of all commands, we introduce an ordered variant of DBFT whose reliable broadcast is replaced by the ordered reliable broadcast and where partial synchrony [7] is assumed. Because all the processes will not receive a command at the same time, they may observe different timestamps, and no particular timestamp is more meaningful than the others. To simplify the agreement process, we introduce an *ordering clock* with a coarser grain than the process clock. Instead of agreeing on a timestamp coming from a process clock, processes will agree on an ordering indicator coming from the ordering clock.

Ordering Clock. Each unit of the ordering clock lasts χ units on the process clock. Each timestamp t coming from a process clock can be mapped to an ordering indicator $\mathsf{order}(t) = \tau$ on the ordering clock, with $\tau\chi \leq t < (\tau + 1)\chi$. When the value of χ is greater than the message propagation time Δ_{GST}, any command c is broadcast and received in a period smaller than χ. If c is broadcast at a time τ on the ordering clock, then either *(i)* c is sent and received in the same unit τ, or *(ii)* c is received by other processes during the next unit $\tau + 1$ (if c was broadcast toward the end of the unit τ). Figure 1 shows examples of how processes may adopt a value on the ordering clock.

Fast Ordered Byzantine Consensus. The goal of the *order agreement* algorithm presented in this section is to decide an ordering indicator from the ordering clock for each command. It requires that each command is broadcast with a timestamp metadata t whose ordering indicator $\tau = \mathsf{order}(t)$ will be used as a *reference order.* During synchronous periods, a command is broadcast and received either during the same unit of ordering time (i.e., at $\tau + 0$), or during the next one (i.e., at $\tau + 1$). During asynchronous periods, the command may be received after a number of ordering units $k > 1$. Deciding a unique ordering indicator for a command can thus be reduced to deciding on a value $k \geq 0$ resulting in an ordering indicator $\tau + k$ (where τ is the reference order of the command). To agree on a value of k that is correctly bounded, processes execute successive rounds of binary consensus, starting with round 0. If the binary consensus instance of round r outputs 1, then the decided ordering indicator is $\tau + r$. The protocol is presented in Algorithm 1. After global stabilization time, and provided that $\chi > \Delta_{GST}$, the decided ordering indicator is either 0 or 1. Thus the protocol first executes these two instances concurrently (line 2). When both of these instances have decided, if one of them has output 1, then the ordering indicator is decided (line 4 or 6). Otherwise, processes will iteratively try to agree on a higher ordering indicator with the loop starting at line 8. During each iteration of the loop, processes first try to output 1 for the current round number, and then try to backtrack (cf. Backtracking). Whenever an ordering indicator is decided, either at line 11 or 14, the algorithm terminates.

▶ **Theorem 1** (Ordering Linearizability). *The order agreement protocol is a distributed ordering algorithm that ensures ordering linearizability with respect to the ordering clock. If we define T_1 (resp. T_2) being the set of timestamps perceived by correct processes for command c_1 (resp. c_2). Then, $\forall t \in T_1, u \in T_2, \mathsf{order}(t) < \mathsf{order}(u) \Rightarrow c_1 \prec c_2$, where $c_1 \prec c_2$ indicates that c_1 executes before c_2 at all correct processes.*

Algorithm 1 Order Agreement.

```
1:  order-agreement($c, T$):
2:    decide-round($c, 0, T$) $\rightarrow$ decided-0 || decide-round($c, 1, T$) $\rightarrow$ decided-1    ▷ execute concurrently
3:    if decided-0 then
4:      return 0                                                ▷ decide 0 as ordering indicator
5:    else if decided-1 then
6:      return 1                                                ▷ decide 1 as ordering indicator
7:    $r \leftarrow 2$                                           ▷ start with round 2
8:    loop:
9:      decide-round($c, r, T$) $\rightarrow$ decided-r          ▷ binary consensus to adopt $r$ as ordering indicator
10:     if decided-r then
11:       return $r$                                            ▷ ordering indicator $r$ decided
12:     decide-backtrack($c, r, T$) $\rightarrow$ backtrack-order    ▷ can a lower ordering indicator be decided
13:     if backtrack-order $\neq \bot$ then
14:       return backtrack-order                                ▷ backtrack decided
15:     $r \leftarrow r + 1$                                     ▷ increment the round number
```

Backtracking. A network adversary could prevent correct processes from reaching agreement, until the round number goes beyond a value that would result in an ordering indicator that would be correctly bounded. The backtracking mechanism enables processes to decide an ordering indicator that is lower than the current round number. This is done by a rotating coordinator that proposes a lower ordering indicator, justified by a set of $2f + 1$ signed timestamps. Processes then execute an instance of binary consensus to decide whether the value of the coordinator can be adopted. If this binary consensus outputs 1, then the value of the coordinator is adopted, and the backtrack agreement returns the value of the coordinator at line 12.

Application to Blockchains. A blockchain [10] is a ledger consisting of a totally ordered set of transactions organized in a chain of blocks. The Red Belly Blockchain [5] ensures *censorship-resistance*, a notion of fairness different from Kelkar et al.'s [8] that ensures that a transaction submitted by a correct process gets eventually executed, however, it does not impose that two transactions perceived in a specific order by all correct processes are executed in the same order. In particular, for each block, processes carry an instance of binary consensus on the transaction proposal of each process, so that the decided block is a subset of the transactions proposed by all processes. Our order agreement algorithm can be used to sequence transaction proposals in each block, where instead of executing the decided transaction proposals in a lexicographical order, proposals are sequenced using a decided ordering indicator. Concurrently to the binary consensus to decide whether a proposal is included in a block, we execute the order agreement to decide an ordering indicator for the proposal. In the fast path, after 6 message delays, both the agreement on the inclusion of the proposal in the block, and the agreement on its ordering indicator have terminated. This parallelism is key to speedup the alternatives of executing a pre-protocol before a consensus [12] or an atomic broadcast [9].

References

1 Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazic, Pierre Tholoniat, and Josef Widder. Compositional Verification of Byzantine Consensus. Technical Report hal-03158911, HAL, March 2021. URL: `https://hal.archives-ouvertes.fr/hal-03158911/file/paper.pdf`.

2 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

3 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, 2020.

4 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its applications to blockchains. In *Proceedings of the IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018.

5 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A secure, fair and scalable open blockchain. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, pages 466–483, 2021.

6 Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (S&P)*, pages 910–927, 2020.

7 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):pp.288–323, 1988.

8 Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for Byzantine consensus. In *Annual International Cryptology Conference (CRYPTO)*, pages 451–480, 2020.

9 Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies (AFT)*, pages 25–36, 2020.

10 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

11 Christopher Natoli and Vincent Gramoli. The blockchain anomaly. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 310–317, 2016.

12 Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 633–649, 2020.